```
Set     Items   Description
S1     164913   PREPROCESSOR? ? OR PRE()PROCESSOR? ? OR PREPROCESSER? ? OR
              PRE()PROCESSER? ? OR PRAGMA OR DIRECTIVE OR METAPROGRAMM??? OR
               COMPILER? ? OR INTERPRETER? ? OR PRECOMPILER? ? OR PROBE
S2   11193522   INSERT??? OR ADD??????? OR EMBED???? OR INCLUD??? OR INCLU-
              S??? OR INTEGRAT??? OR IMPLEMENT????? OR AUGMENT????? OR UPDA-
              T? OR UP()(DATE? OR DATING? OR GRAD?) OR UPGRAD? OR GENERAT???
               OR EXTEND??? OR EXTENS??? OR INCORPORAT???
S3    3542429   CONSTRUCT? ? OR FUNCTION? ? OR DATA()TYPE? ? OR LANGUAGE()-
              ELEMENT? ? OR OPERAND? ? OR OPERATOR? ? OR OPERATION? ? OR CO-
              NTROL()STRUCTURE? ? OR STRUCTURE()DEFINITION? ? OR SYMBOL? ? -
              OR BOOLEAN? ? OR HIGH()LEVEL()LANGUAGE? ? OR ROUTINE? ? OR SU-
              BROUTINE? ? OR
S4    2874277   EAS??? OR FACILITAT??? OR SIMPLIF???????
S5       4384   S1 AND S2 AND S3 AND S4
S6        863   ASL OR ADVANCED(2W)CONFIGURATION(2W)POWER(2W)INTERFACE OR -
              ACPI OR AML
S7          1   S5 AND S6
S8         84   AU=(QURESHI S? OR QURESHI, S?)
S9         19   S8 AND S6
S10       242   ((S5 OR S6) AND (MC=T01 OR IC=(G06F-009/45 OR G06F-001/26)-
              )) NOT (S7 OR S9 OR AD=(20021023:20051023) OR AD=(20051023:20-
              060809))
S11        40   S10 AND (POWER OR ACPI OR ASL OR AML)
S12        84   (S10 AND (S2(5N)S3)) NOT S11
? show files
File 347:JAPIO Dec 1976-2005/Dec(Updated 060404)
         (c) 2006 JPO & JAPIO
File 350:Derwent WPIX 1963-2006/UD=200650
         (c) 2006 The Thomson Corporation
?
```

```
50406  CONSTRUCT? ?
        954934  FUNCTION? ?
       1889146  DATA
       2294963  TYPE? ?
          3690  DATA(W)TYPE? ?
        800149  LANGUAGE
       2304659  ELEMENT? ?
           138  LANGUAGE(W)ELEMENT? ?
          8721  OPERAND? ?
        230375  OPERATOR? ?
       2426718  OPERATION? ?
       3709694  CONTROL
       2021895  STRUCTURE? ?
          4025  CONTROL(W)STRUCTURE? ?
       1920271  STRUCTURE
         90383  DEFINITION? ?
           318  STRUCTURE(W)DEFINITION? ?
         75929  SYMBOL? ?
          2005  BOOLEAN? ?
       4294666  HIGH
        960783  LEVEL
        801975  LANGUAGE? ?
           845  HIGH(W)LEVEL(W)LANGUAGE? ?
         29167  ROUTINE? ?
          3198  SUBROUTINE? ?
        779768  SUB
         29167  ROUTINE? ?
           901  SUB(W)ROUTINE? ?
         16640  MACRO? ?
         38726  LIMITATION? ?
         61866  RULE? ?
    S3 3542429  CONSTRUCT? ? OR FUNCTION? ? OR DATA()TYPE? ? OR
                LANGUAGE()ELEMENT? ? OR OPERAND? ? OR OPERATOR? ? OR
                OPERATION? ? OR CONTROL()STRUCTURE? ? OR
                STRUCTURE()DEFINITION? ? OR SYMBOL? ? OR BOOLEAN? ? OR
                HIGH()LEVEL()LANGUAGE? ? OR ROUTINE? ? OR SUBROUTINE? ?
                OR SUB()ROUTINE? ? OR MACRO? ? OR LIMITATION? ? OR RULE?
                ?
```

9/5/1 (Item 1 from file: 350)
DIALOG(R)File 350:Derwent WPIX


0014996810 - Drawing available
WPI ACC NO: 2005-344694/200535
XRPX Acc No: N2005-281654
Advanced configuration and power interface source language code compiling method in computer system, involves executing pre-processor to process ASL program and to insert support for one non-native programming construct
Patent Assignee: QURESHI S A (QURE-I)
Inventor: QURESHI S A
Patent Family (1 patents, 1 countries)

| Patent Number | Kind | Date | Application Number | Kind | Date | Update |
|---|---|---|---|---|---|---|
| US 20050091649 | A1 | 20050428 | US 2003693510 | A | 20031024 | 200535 B |

Priority Applications (no., kind, date): US 2003693510 A 20031024

Patent Details

| Number | Kind | Lan | Pg | Dwg | Filing Notes |
|---|---|---|---|---|---|
| US 20050091649 | A1 | EN | 10 | 4 | |

Alerting Abstract US A1
NOVELTY - The advanced configuration and power interface ( ACPI ) source language ( ASL ) program is accessed and a pre-processor is executed to process ASL program and to insert support for one non-native programming construct. The ASL program is then complied to advanced configuration and power interface machine language (AML) program.
DESCRIPTION - INDEPENDENT CLAIMS are also included for the following:

1. computer-readable medium storing advanced configuration and power interface source language code compiling program; and

2. system for compiling advanced configuration and power interface source language code into machine language code.

USE - For compiling advanced configuration and power interface ( ACPI ) source language ( ASL ) code into advanced configuration and power interface machine language (AML) code in digital computer system.
ADVANTAGE - Enables compiling the advanced configuration and power interface source language ( ASL ) code into advanced configuration and power interface machine language code, within short period and at low cost.
DESCRIPTION OF DRAWINGS - The figure shows the flowchart illustrating the compiling process of ASL code.

Title Terms/Index Terms/Additional Words: ADVANCE; CONFIGURATION; POWER; INTERFACE; SOURCE; LANGUAGE; CODE; COMPILE; METHOD; COMPUTER; SYSTEM; EXECUTE; PRE; PROCESSOR; PROCESS; PROGRAM; INSERT; SUPPORT; ONE; NON; NATIVE; CONSTRUCTION

Class Codes
International Classification (Main): G06F-009/45
.(Additional/Secondary): G06F-001/26
US Classification, Issued: 717140000, 713300000

File Segment: EPI;
DWPI Class: T01
Manual Codes (EPI/S-X): T01-F05A; T01-S03

US006298389B1

(12) **United States Patent**
**Gerhardt**

(10) **Patent No.: US 6,298,389 B1**
(45) **Date of Patent: Oct. 2, 2001**

(54) **METHOD FOR INPUT AND OUTPUT OF STRUCTURES FOR THE JAVA LANGUAGE**

(75) Inventor: **Alan L. Gerhardt**, Dallas, TX (US)

(73) Assignee: **Compaq Computers, Inc.**, Cupertino, CA (US)

( * ) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **08/879,737**

(22) Filed: **Jun. 20, 1997**

(51) **Int. Cl.**[7] ..... **G06F 9/45**; G06F 9/54
(52) **U.S. Cl.** ..... **709/313**; 717/5
(58) **Field of Search** ..... 395/705, 706, 395/707, 680, 703, 702; 709/300

(56) **References Cited**

U.S. PATENT DOCUMENTS

| | | | | |
|---|---|---|---|---|
| 5,361,350 | * | 11/1994 | Conner et al. | 395/600 |
| 5,493,680 | * | 2/1996 | Danforth | 395/700 |
| 5,675,801 | * | 10/1997 | Lindsey | 395/702 |
| 5,732,257 | * | 3/1998 | Atkinson et al. | 395/604 |
| 5,884,083 | * | 3/1999 | Royce et al. | 395/705 |

OTHER PUBLICATIONS

(Digital Cat) "C2J, A C + + to Java translator". www.javacats.com/US/search/rid00000279.html, Sep. 20, 1996.*
Jaworski, Jamie. "JAVA Developer's Guide", 1996.*
Ber Elliot Joel. "Java–Lex: A lexical analyzer generator for Java", Aug. 15, 1996.*
Gibello, Pierre–Yves. "ZQL: a Java SQL parser", Mar. 1998.*
Laffra, Chris. "Advanced Java". pp. 253–262 Sep. 1996.*
Aho, Alfred V., Ravi Sethi, Jeffrey D. Ullman. "Compilers: Principles, Techniques, and Tools". Chap. 9, Code Generation; p. 513–528, 1986.*
Kochan, Stephen G.. "Programming in C, Revised Edition". pp. 369–371, 1988.*
"Orbix for Java White Paper". IONA Technologies Ltd., p. 1–14, Feb. 1996.*
Eckel, Bruce. "C + +Idiom", pp. 1–15, Jul. 1993.*

* cited by examiner

*Primary Examiner*—Alvin Oberley
*Assistant Examiner*—Lewis A. Bullock, Jr.
(74) *Attorney, Agent, or Firm*—Fenwick & West LLP

(57) **ABSTRACT**

The present invention includes a method and apparatus that allows languages to send, receive and manipulate structures defined by other languages. Structurally, the present invention includes a preprocessor and a runtime library. The preprocessor accepts, as input, source code written in a high-level language, such as C or C++. The preprocessor produces, as output, a series of Java classes. Each Java class describes a structure defined in the input source code. Java programs use the descriptions produced by the preprocessor to send, receive and manipulate structures the structures defined in the input source code.

**19 Claims, 5 Drawing Sheets**

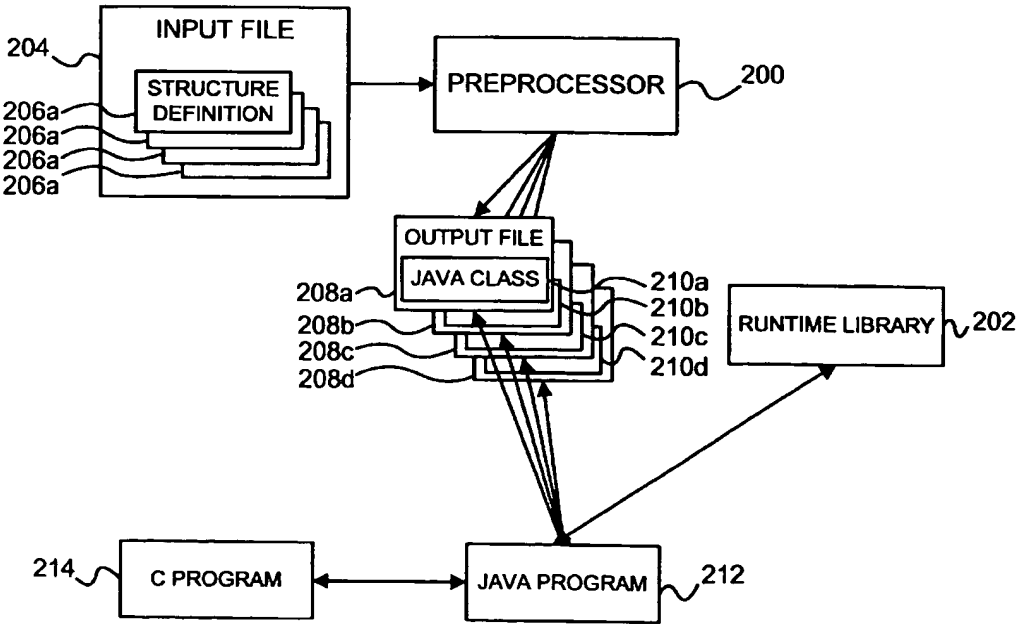


Fig. 1

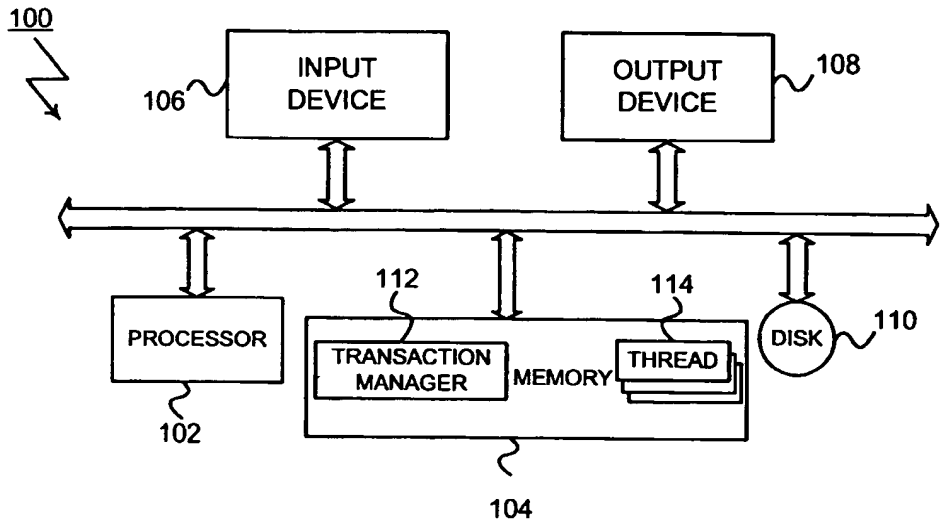

100
INPUT DEVICE
106
OUTPUT DEVICE
108
PROCESSOR
102
112
TRANSACTION MANAGER
MEMORY
114
THREAD
DISK
110
104

Fig. 2

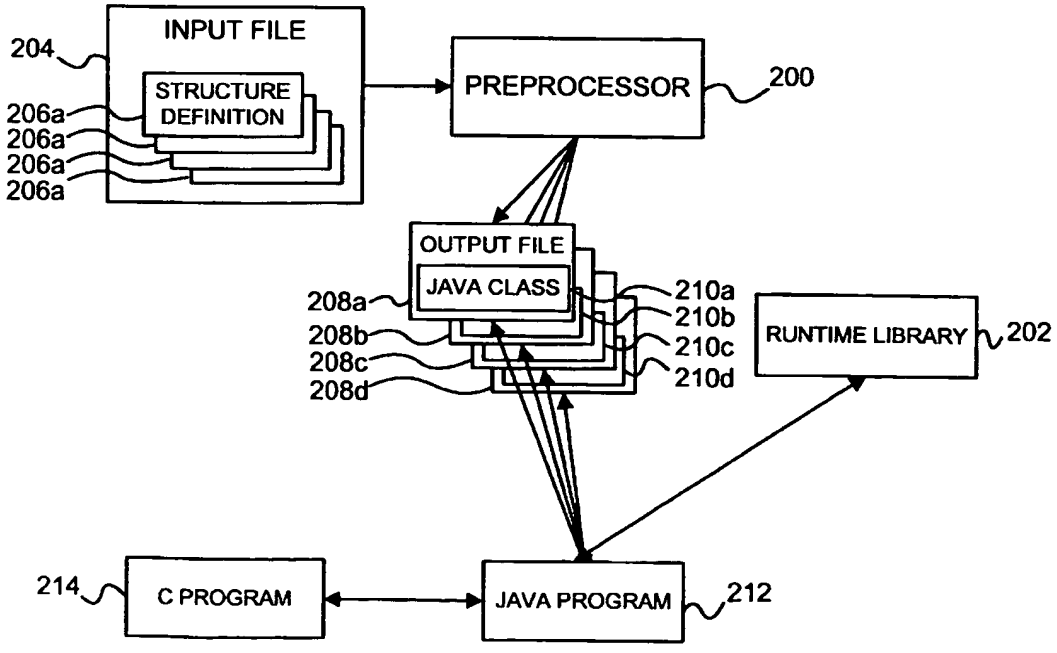

INPUT FILE
204
STRUCTURE DEFINITION
206a
206a
206a
206a
PREPROCESSOR
200
OUTPUT FILE
JAVA CLASS
208a
208b
208c
208d
210a
210b
210c
210d
RUNTIME LIBRARY
202
214
C PROGRAM
JAVA PROGRAM
212

Fig. 3

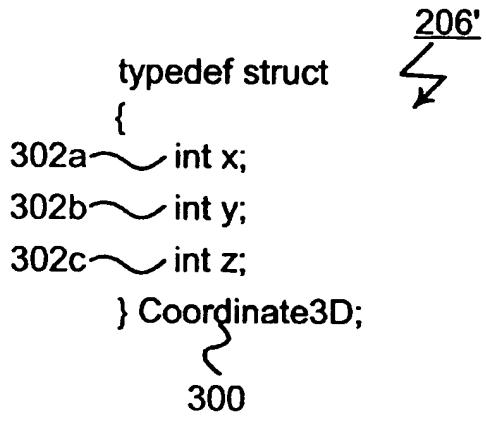

206'
typedef struct
{
302a int x;
302b int y;
302c int z;
} Coordinate3D;
300

Fig. 4

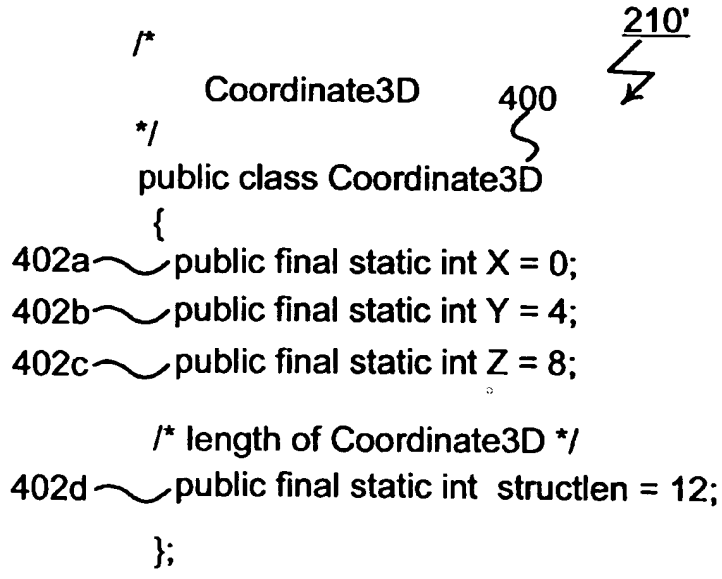

210'
/*
Coordinate3D
400
*/
public class Coordinate3D
{
402a public final static int X = 0;
402b public final static int Y = 4;
402c public final static int Z = 8;
/* length of Coordinate3D */
402d public final static int structlen = 12;
};

Fig. 5

206"

```
typedef struct
{
500a~ Coordinate3D Start;
500b~ Coordinate3D Stop;
} Trip;
```

Fig. 6

210"

```
/*
    Trip
*/
public class Trip
 {
600a~public final static int  Start$X = 0;
600b~public final static int  Start$Y = 4;
600c~public final static int  Start$Z = 8;
600d~public final static int  Stop$X = 12;
600e~public final static int  Stop$Y = 16;
600f~public final static int  Stop$Z = 20;

 /* length of Trip */
600g~public final static int  structlen = 24;
 };
```

Fig. 7

206'''

```
        typedef struct
        {
700a    Coordinate3D Start[10];
700b    Coordinate3D Stop[5];
        } Trip;
```

Fig. 8

210'''

```
        /*
            Trip
        */
        public class Trip
         {
800a    public final static int  Start$X[10] = {0, 12, 24, 36, 48, 60, 72, 84, 96, 108};
800b    public final static int  Start$Y[10] = {4, 16, 28, 40, 52, 64, 76, 88, 100, 112};
800c    public final static int  Start$Z[10] = {8, 20, 32, 44, 56, 68, 80, 92, 104, 116};
800d    public final static int  Stop$X[5] = {120, 132, 144, 156, 168};
800e    public final static int  Stop$Y[5] = {124, 136, 148, 160, 172};
800f    public final static int  Stop$Z[5] = {128, 140, 152, 164, 176};

        /* length of Trip */
800g    public final static int  structlen = 180;
         };
```

Fig. 9

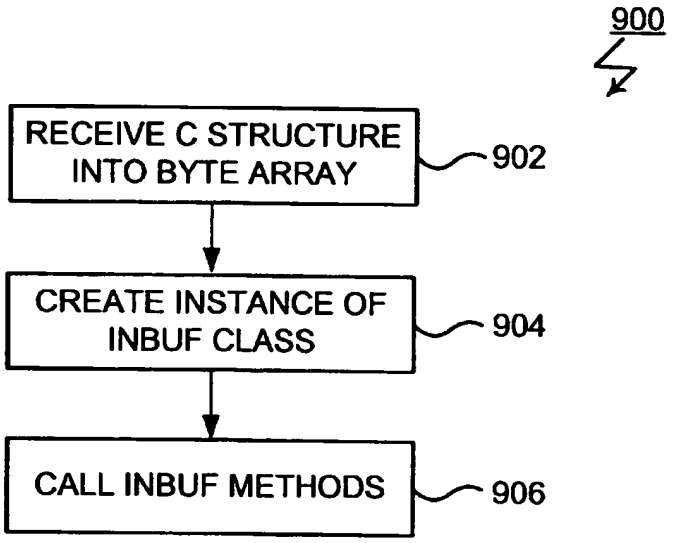



Fig. 10

```
1002 ~ byte [] buffer = new byte[Coordinate3D.structlen];
1004 ~ in.read (buffer, 0, Coordinate3D.structlen);
1006 ~ inbuf received_structure = new inbuf (buffer);

1008 ~ int X = received_structure.readInt (Coordinate3D.X);
1010 ~ int Y = received_structure.readInt (Coordinate3D.Y);
1012 ~ int Z = received_structure.readInt (Coordinate3D.Z);
```

1000

# METHOD FOR INPUT AND OUTPUT OF STRUCTURES FOR THE JAVA LANGUAGE

## FIELD OF THE INVENTION

The present invention relates generally to interprocess communication between processes created that use different programming languages. More specifically, the present invention includes a method and apparatus for input, output and manipulation of structures within processes created using languages that do not support structures.

## BACKGROUND OF THE INVENTION

Interprocess communication, or IPC, is a general term for techniques that allow software processes to exchange information. By exchanging information, software processes cooperate to perform tasks. Functionally, the use of IPC offers a number of advantages over more traditional monolithic programming models. These advantages include ease of implementation, component reusability, and encapsulation of information. This combination of advantages has resulted in the widespread use of IPC.

Over time, IPC techniques have been adapted to meet the demands of increasingly sophisticated programming environments. Many of these adaptations have attempted to make IPC techniques easier for programmers to use. In spite of this evolution, there are still cases where programmers find the use of IPC techniques to be difficult or error prone. One of these cases arises when programmers attempt to use IPC techniques to send and receive structures within processes created using programming languages that do not provide support for structures. In more detail, high-level languages, such as C, C++ or Pascal, typically allow programmers to group data into aggregates known as structures. In practice, programmers typically use structures to compartmentalize or group related data. This allows related data to be manipulated as a single entity. Thus, for example, a structure might include data describing a hardware device, or data describing an employee record. Because structures contain related data, it is natural for cooperating processes to exchange structures using IPC.

Some high-level languages, namely Java, do not allow data to be grouped into structures. As a result, programmers find it difficult to program Java processes to receive and manipulate structures sent by other processes using IPC. The lack of support for structures also makes it difficult to program Java processes to send structures to other processes using IPC. The unfortunate result is that it is difficult to construct systems where Java-based processes cooperate with processes implemented using more traditional languages, such as C or C++. Since heterogeneous systems of this type are often useful, there exists a need for languages, like Java, to send, receive and manipulate structures.

## SUMMARY OF THE INVENTION

The present invention includes a method and apparatus that allows languages to send, receive and manipulate structures defined by other languages. Structurally, the present invention includes a preprocessor and a runtime library. The preprocessor accepts, as input, source code written in a high-level language, such as C or C++. The preprocessor produces, as output, a series of Java source code modules.

Each Java source module output by the preprocessor includes a Java class. Each class corresponds to one of the structures defined in the preprocessor input. The preprocessor maintains this correspondence by naming each class to match the name of the corresponding structure definition. Each class functions as a description of the structure that it is named after. To perform this function, each class includes a series of static integers. One of these static integers is preferably given the reserved name structlen. The structlen static integer included in a class is initialized to be equal to the size of structure that the class is named after. Each of the remaining static integers included in a class is named after one of the elements included in the structure that the class is named after. Each of these static integers is initialized to equal the byte offset of the structure element that it is named after.



The classes are emitted by the preprocessor function as descriptions of the structures included in the C or C++ input file. As classes, these descriptions are available within Java programs. In this way, the present invention provides a method for generating descriptions of C and C++ structures with descriptions available within Java programs. Importantly, the descriptions provided by the preprocessor include the size of each structure and the offset of each element within the structures. These values are available as constants and may be used by programmers creating Java programs.

The present invention also includes a runtime library. The runtime library includes an inbuf Java class. The inbuf class includes a series of methods each designed to read a built-in Java type, such as an integer or character, from a byte array. Each method accepts an argument telling the method the offset within the byte array that the read of the built-in type is to be performed.

Programmers create Java programs to receive, manipulate and send C++ structures using the runtime library and the classes provided by the preprocessor. To receive a C or C++ structure, a programmer configures a Java program to call a standard Java IPC method. The received structure is then stored as a byte array. To access a structure element within the byte array, the programmer configures the Java program to call the appropriate inbuf method. Thus, if an integer value is to be accessed, the programmer configures the Java program to call the inbuf method for reading integers. The programmer specifies the offset of the desired element within the byte array using the description of the element included in the appropriate class provided by the preprocessor.

Advantages of the invention will be set forth, in part, in the description that follows and, in part, will be understood by those skilled in the art from the description or may be learned by practice of the invention. The advantages of the invention will be realized and attained by means of the elements and combinations particularly pointed out in the appended claims and equivalents.

## BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, that are incorporated in and constitute a part of this specification, illustrate several embodiments of the invention and together with the description, serve to explain the principles of the invention.

FIG. 1 is a block diagram of a host computer system in accordance with a preferred embodiment of the present invention.

FIG. 2 is a block diagram showing the components included in a preferred embodiment of the present invention.

FIG. 3 is a diagram of an exemplary C structure definition.

FIG. 4 is a diagram of a Java class created by the preprocessor of the present invention for the structure definition of FIG. 3.

FIG. 5 is a diagram of an exemplary nested C structure definition.

FIG. 6 is a diagram of a Java class created by the preprocessor of the present invention for the nested structure definition of FIG. 5.

FIG. 7 is a diagram of an exemplary C structure definition that includes nested arrays of structures.

FIG. 8 is a diagram of a Java class created by the preprocessor of the present invention for the structure definition of FIG. 7.

FIG. 9 is a flowchart showing the steps performed by a Java program to receive and manipulate a C structure using the Java classes and runtime library included in a preferred embodiment of the present invention.

FIG. 10 is a Java code fragment that corresponds to the method of FIG. 9.

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Reference will now be made in detail to preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. Wherever possible, the same reference numbers will be used throughout the drawings to refer to the same or like parts.

### Environment

In FIG. 1, a host computer system **100** is shown as a representative environment for the present invention. Structurally, host computer system **100** includes a processor, or processors **102**, and a memory **104**. An input device **106** and an output device **108** are connected to processor **102** and memory **104**. Input device **106** and output device **108** represent a wide range of varying I/O devices such as disk drives, keyboards, modems, network adapters, printers and displays. Host computer system **100** also includes a disk drive **110** of any suitable disk drive type (equivalently, disk drive **110** may be any non-volatile storage system such as "flash" memory). A transaction manager **112**, and one or more threads **114** are shown to be resident in memory **104** of host computer **100**.

### Overview

Turning to FIG. 2 it may be seen that the present invention includes a preprocessor **200** and a runtime library **202**. Preprocessor **200** operates on input file **204**. Input file **204** contains C or C++ source code and includes a series of structure definitions **206**. For each structure definition **206** included in input file **204**, preprocessor **200** produces a respective output file **208**. Each output file **208** contains Java source code that defines a respective Java class **210**. Java classes **210** act as descriptions of the structures defined by structure definitions **206**. Programmers create Java programs, such as Java program **212**, that use the descriptions provided by Java classes **210**. Together, the use of these descriptions, along with calls to runtime library **202**, allows Java program **212** to receive and manipulate structures sent by C or C++ programs, such as C program **214**. Importantly, it will generally be the case that input file **204** will be one of the files used to construct C program **214**. In this way, C program **214** and Java program **212** automatically use the same definitions for equivalent structures.

### Preprocessor

Preprocessor **200** is preferably implemented as a parser that accepts the C and C++ languages. In general, parsers of this type may be constructed using a range of generally available parser generation tools, such as YACC or Bison. The design and construction of parsers is described in detail in Aho, Sethi and Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley (1986), Chap. 1–5, pp. 1–342, the disclosure of which is incorporated into this document by reference. As a parser, preprocessor **200** recognizes semantic constructs within input file **204**. For most semantic constructs, preprocessor **200** is configured to produce no output. Thus, the default translation performed by preprocessor **200** is no output. The default translation is not used, however, when preprocessor **200** recognizes semantic constructs that correspond to structure definitions **206**. Instead, preprocessor **200** produces an output file **208** containing a Java class **210** for each structure definition **206** recognized in input file **204**.

The relation of structure definitions **206** to Java classes **210** is better understood by reference to structure definition **206'** of FIG. 3 and corresponding Java class **210'** of FIG. 4. As shown, structure definition **206'** has a structure name **300** which, for purposes of illustration, is shown as the name "Coordinate3D." Structure definition **206'** also includes three integer elements **302***a* through **302***c* that are shown as X, Y and Z, respectively. In general, it is to be appreciated that structure definition **206'** is intended to be exemplary of the type of structure definitions **206** that may be included in any C or C++ source code file, such as input file **204**.

FIG. 4 shows the Java class **210'** output by preprocessor **200** for the particular structure definition **206'**. Java class **210'** has a class name **400** that corresponds to structure name **300** (i.e., both class name **400** and structure name **300** are the name "Coordinate3D"). Java class **210'** includes four class members **402***a* through **402***d*. Each class member **402** is a public final static integer. Each of the first three of class members **402** corresponds to an element **302** included in structure definition **206'** of FIG. 3. Thus, class member **402***a* corresponds to structure element **302***a*, class member **402***b* corresponds to structure element **302***b* and class member **402***c* corresponds to structure element **302***c*. Preprocessor **200** establishes this correspondence by naming each class member **402** to match the corresponding structure element **302**. Thus, class member **402***a* and structure element **302***a* are both named "X." As described, preprocessor **200** names Java class **210'** and class members **402** to match the corresponding structure name **300** and structure elements **302**. It should be appreciated, however, that alternate naming schemes may be used without deviating from the spirit of the present invention. In particular, there may be cases where mangled names (such as names including an initial underscore) may be more appropriate.

Preprocessor **200** initializes each of the first three of class members **402** to describe the corresponding structure element **302**. In particular, each of these class members **402** is initialized to be equal to the byte offset of the corresponding structure element **302** within structure definition **206'**. Thus, class member **402***b* is initialized to the value of four since structure element **302***b* is positioned four bytes from the start of structure definition **206'**.

Unlike class members **402***a* through **402***c*, class member **402***d* does not correspond to a structure element **302**. Instead, preprocessor **200** initializes class member **402***d* to be equal to the total size, in bytes, of the structure defined by structure definition **206'**. In the case of structure definition **206'** of FIG. 3 the defined structure includes three four-byte integers for a total of twelve bytes. Thus, for this example, class member **402***d* is initialized by preprocessor **200** to have the value twelve.

FIGS. 3 and 4 show the correspondence between structure definition **206'** and Java class **210'**. It may be appreciated, however, that structure definition **206** is relatively simple and includes no nested structures. The use of preprocessor

200 to create Java classes 210 for more generalized structure definitions is better understood by reference to structure definition 206" of FIG. 5 and corresponding Java class 210" of FIG. 6. As shown in FIG. 5, structure definition 206" includes two nested structures 500*a* and 500*b*. Each of these nested structures 500 is an instance of the Coordinate3D structure of structure definition 206' of FIG. 3. As a result, structure definition 206" includes a total of six elements. These structure elements are: Start.X, Start.Y, Start.Z, Stop.X, Stop.Y and Stop.Z.

FIG. 6 shows the Java class 210" constructed by preprocessor 200 for structure definition 206". Structure definition 206" includes seven class members 600*a* through 600*g*. Each class member 600 is a public final static integer. Each class member 600 corresponds to one of the structure elements included in structure definition 206". Preprocessor 200 establishes this correspondence by naming each class member 600 to match the corresponding structure element. Thus, class member 600*a* is named "Start$X, class member 600*b* is named Start$Y and so on.

Preprocessor 200 initializes class members 600 in the same fashion as described for structure definition 206' and Java class 210'. Thus, preprocessor 200 initializes each of the first six class members 600 to be equal to the byte offset of the corresponding structure element within structure definition 206". In the case of class member 600*a*, this value is zero. For class member 600*b*, this value is four. The value increases by four for each of the first six class members 600.

Another example of the use of preprocessor 200 to create Java classes 210 for structure definitions is better understood by reference to structure definition 206'" of FIG. 7 and corresponding Java class 210'" of FIG. 8. As shown in FIG. 7, structure definition 206'" includes two nested arrays of structures 700*a* and 700*b*. Array of structures 700*a*includes ten instances of the Coordinate3D structure (i.e., the structure definition 206' of FIG. 3). Array of structures 702*b* is similar, except that five instances of the Coordinate3D structure are included. Overall, structure definition 206'" includes fifteen instances of the Coordinate3D structure for a total of forty-five elements. These structure elements are: Start[0].X, Start[0].Y, Start[0].Z . . . Start[9].X, Start[9].Y, Start[9].Z and Stop[0].X, Stop[0].Y and Stop[0].Z . . . Stop[4].X, Stop[4].Y and Stop[4].Z.

FIG. 8 shows the Java class 210'" constructed by preprocessor 200 for structure definition 206'". Structure definition 206'" includes seven class members 800*a* through 800*g*. The first six class members 800 are public final arrays of static integers. Each of these class members 800 corresponds to one of the structure elements included in array of structures 702*a* or array of structures 702*b*. For example, class member 800*a* corresponds to the element X included in array of structures 700*a*. Similarly, class members 800*e* corresponds to the element Y included in array of structures 700*b*. Preprocessor 200 establishes this correspondence by declaring each class member 800 to match the corresponding structure element. Thus, class member 800*a* is named "Start$X, class member 800*b* is named Start$Y, and so on.

Preprocessor 200 initializes each class member 800 to be equal to the byte offset of the corresponding structure element within structure definition 206'". This is accomplished by emitting a series of offsets for each class member 800. For example, the offsets 0, 12, 24, 36, 48, 60, 72, 84, 96 and 108 are emitted for class member 800*a*. The offsets 4, 16, 28, 40, 52, 64, 76, 88, 100 and 112 are emitted for class member 800*b*.

Preprocessor 200 constructs a Java class 210 of the type shown in FIGS. 4, 6 and 8 for each structure definition 206 included in input file 204. Together, the Java classes 210 output by the preprocessor 200 form a complete description of the structure definitions 206 included in input file 204.

Runtime Library

Runtime library 202 is preferably implemented to include a Java class inbuf. The inbuf class is declared as follows:

```
one public class inbuf
{
public inbuf (byte[ ] byte_array);
public String readString (int offset, int strlen);
public byte readByte (int offset);
public char readChar (int offset);
public short readShort (int offset);
public int readInt (int offset);
public long readLong (int offset);
public byte[ ] readData (int offset);
}
```

As shown, the inbuf class includes an inbuf constructor that takes, as an argument, a byte array. The remaining methods included in the inbuf class read and return intrinsic Java types, such as Strings, bytes, chars and shorts. Each method for reading an intrinsic type takes an integer offset argument. Each method reads the appropriate intrinsic type from the byte array passed to the inbuf constructor at the offset passed to the method. Thus, the inbuf class provides a generalized set of methods for reading intrinsic types from byte arrays and allows the offset for reading within the byte array to be specified.

Internally, the inbuf method is preferably implemented using the standard Java.io package. The Java.io package includes a number of classes including a ByteArrayInputStream class and a DataInputStream class. To use the Java.io package, the inbuf constructor creates a ByteArrayInputStream using the byte array passed to the inbuf constructor. The inbuf constructor then creates a DataInputStream using the created ByteArrayInputStream. The DataInputStream provides the methods that are used by the inbuf class to read and return intrinsic types from the byte array passed to the inbuf constructor.

Together, Java classes 210 emitted by preprocessor 200 and runtime library 202 provide a generalized method for manipulating C and C++ structures within Java programs. An exemplary use of Java classes 210 and runtime library 202 is shown as method 900 of FIG. 9. Method 900 begins with step 902 where Java program 212 receives a C structure from C program 214. In general, it should be appreciated that step 902 may be performed using a wide range of IPC techniques and will generally require a combination of IPC routines called by C program 214 and Java program 212. In this way, step 902 corresponds to the use of these techniques to transfer the data included in a structure from C program 214 to Java program 212. At the completion of step 902, the received structure is stored in a Java byte array.

In step 904, Java program 212 creates an instance of the inbuf class for the newly received structure. Creation of the inbuf class is performed by calling the inbuf constructor, passing the byte array where the received structure is stored, as an argument.

In step 906, Java program 214 reads an element of the received structure. To perform this read, Java program 214 calls a method of the inbuf class instance created in step 906. To specify the offset within the byte array where the read will be performed, Java program 214 passes the class member 402 that corresponds to the desired structure element 302.

In FIG. 10, a fragment of Java code illustrating the use of method 900 is shown and is generally designated 1000. At line 1002, Java fragment 1000 allocates a byte array using the Java new constructor. Since the allocated byte array is intended to receive a structure of the type defined by structure definition 206' of FIG. 3, the value

Coordinate3D.structlen (i.e., a value of 12) is passed to the new constructor. At line **1004**, the Java I/0 method read is called to receive the desired structure into the allocated byte array. Once again, the value Coordinate3D.structlen is used, this time to inform the read method to read **12** bytes into the allocated byte array. Line **1004** corresponds, in a general sense, to step **902** of method **900**.

At line **1006** of fragment **1000**, a new inbuf is created for the allocated byte array. This line corresponds to step **904** of method **900**. In the following three lines (i.e., lines **1008**, **1010** and **1012**), the inbuf readInt method is called to read the X, Y and Z structure elements **302** from the allocated byte array. In each case, the appropriate class member **402** is passed to the readInt method. Thus, to read the X structure element **302*a***, the class member Coordinate3D.X is passed to readInt. Lines **1008**, **1010** and **1012** correspond to step **906** of method **900**.

The present invention is capable of numerous embodiments. For example, it is entirely possible to create preprocessor **200** as a parser for languages other than C and C++. This allows Java program **212** to use structures defined using languages other than C and C++. Preprocessor **200** may also be configured to emit classes designed for use in languages other than Java. This allows the present invention to be used to pass structures to programs that are implemented using languages like Java that do not support structures. It is also possible for preprocessor **200** to encode additional information within class members **402**. For example, type information might be encoded in class members **402** and then checked at runtime by Java program **212**.

Other embodiments will be apparent to those skilled in the art from consideration of the specification and practice of the invention disclosed herein. It is intended that the specification and examples be considered as exemplary only, with a true scope of the invention being indicated by the following claims and equivalents.

What is claimed is:

**1**. A method for using a structure defined by using a first programming language in a program defined by using a second programming language, the method including the steps, performed by a computer system, of:

receiving a semantic construct that corresponds to a structure definition in the first programming language, the structure definition having a number of elements;

defining a class corresponding to the structure definition using the second programming language; and

defining one or more class members in the class, each class member corresponding to a respective element included in the structure definition, each class member initialized to be a value equal to the byte offset of the corresponding element in the structure definition.

**2**. The method as recited in claim **1** further comprising the step of defining an additional class member which does not correspond with an element in the structure definition but which is initialized to be a value equal to the size of the overall structure definition.

**3**. The method as recited in claim **1** wherein the structure definition has a structure name and wherein the class is defined to have a class name that is identical to the structure name.

**4**. The method as recited in claim **1** wherein each element in the structure definition has an element name and wherein each class member is defined to have a member name that is identical to the corresponding element name.

**5**. A method as recited in claim **1** wherein the class is defined as a Java class.

**6**. A method as recited in claim **1** wherein the structure is defined as a C or C++ structure.

**7**. A preprocessor for receiving programs and structures defined by using a first programming language and producing classes defined using a second programming language, the preprocessor performing the steps of:

receiving semantic constructs and recognizing which semantic constructs correspond with structure definitions defined in the first programming language;

producing class definitions in the second programming language, each respective class definition corresponding to a recognized structure definition;

defining class members for each class definitions, wherein each respective class member corresponds with an element included in the structure definition corresponding to the class definition, each class member being initialized to be a value equal to the offset of the corresponding element described by that class member within the structure definition corresponding to that class definition.

**8**. A preprocessor as recited in claim **7**, wherein each class member included in a class definition is named after the element described by that class member.

**9**. The preprocessor as recited in claim **7**, wherein each class definition is named after the structure definition corresponding to that class definition.

**10**. The preprocessor as recited in claim **7**, wherein each class definition includes an additional class member, whereby the additional class member is initialized to be a value equal to the overall size of the structure definition which corresponds to that class definition.

**11**. The preprocessor as recited in claim **7**, wherein the first programming language is C or C++.

**12**. The preprocessor as recited in claim **7**, wherein the second programming language is Java.

**13**. A computer program product for receiving programs and structures defined by using a first programming language and producing classes defined using a second programming language, the computer program product comprising:

a computer-usable medium having computer-readable code embodied therein for performing the steps of:

recognizing structures defined in the first programming language;

producing class definitions, each respective class definition corresponding to a recognized structure; and

defining class members for each class definition, each respective class member describing an element included in the structure corresponding to the class definition, with each class member included in any one class definition initialized to be a value equal to the byte offset of the element described by that class member within the structure corresponding to that class definition.

**14**. A computer program product as recited in claim **13**, wherein each class member included in a class definition is named after the element described by that class member.

**15**. A computer program product as recited in claim **13**, wherein each class definition is named after the structure corresponding to that class definition.

**16**. The computer program product as recited in claim **13** wherein each class definition includes an additional class member, the additional class member initialized to be a value equal to the size of the structure corresponding to that class definition.

**17**. The computer program product as recited in claim **13** wherein the first programming language is C or C++.

**18**. The computer program product as recited in claim **17** wherein the second programming language is Java.

**19**. The computer program product as recited in claim **15** wherein the class definitions and class member definitions are produced using a second programming language.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

# CERTIFICATE OF CORRECTION

PATENT NO. : 6,298,389 B1
DATED : October 2, 2001
INVENTOR(S) : Alan L. Gerhadt



It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page,
Item [56], U.S. PATENT DOCUMENTS, add the following reference:

-- 5,297,279 * 3/1994 Bannon et al. .....395/600 --

Signed and Sealed this

Fifth Day of August, 2003

JAMES E. ROGAN
*Director of the United States Patent and Trademark Office*

US006195792B1

(12) **United States Patent**
**Turnbull et al.**

(10) **Patent No.: US 6,195,792 B1**
(45) **Date of Patent: Feb. 27, 2001**

(54) **SOFTWARE UPGRADES BY CONVERSION AUTOMATION**

(75) Inventors: **Mark Andrew Turnbull; Harold Joseph Johnson**, both of Nepean (CA)

(73) Assignee: **Nortel Networks Limited**, Montreal (CA)

( * ) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/026,173**

(22) Filed: **Feb. 19, 1998**

(51) **Int. Cl.**[7] ..... **G06F 9/45**
(52) **U.S. Cl.** ..... **717/5**; 717/8; 717/1
(58) **Field of Search** ..... 395/705, 706, 395/704, 708; 717/5, 6, 4, 8, 1

(56) **References Cited**

U.S. PATENT DOCUMENTS

5,581,696 * 12/1996 Kolawa et al. ..... 714/38

OTHER PUBLICATIONS

Farnum, Pattern-based Tree Attribution, ACM 089791-453-8/92/0001/0211, Aug. 1992.*
Aho-Sethi-Ullman, Compiler: Principles, Techniques, and Tools, Chapter 6, Addison-Wesley Publishing Company, Sep. 1985.*
Consel-Danvy, Static and Dynamic Semantics Processing, ACM 089791-419-8/90/0012/0014, Aug. 1990.*
Petterson-Fritzson, DML-A meta-language and System for the Generation of Practical and Efficient Compilers from Denotational Specifications, IEEE 0-8186-2585-Jun. 1992.*
Maranget, Compiling Lazy Pattern Matching, ACM 0-89791-483-X/92/0006/0021, Oct. 1992.*
Lai et al., Architecture Development Environment, IEEETENCON'93, Nov. 1993.*

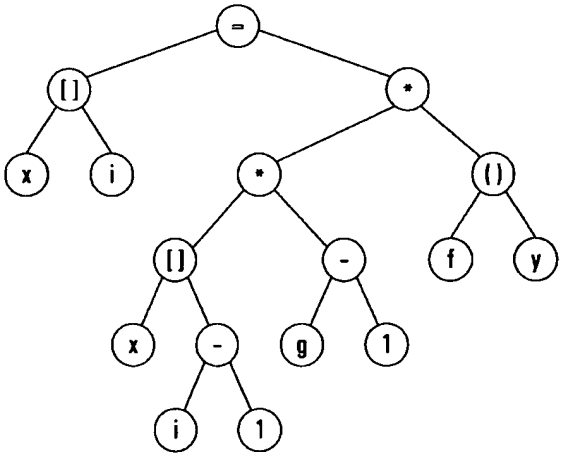
* cited by examiner

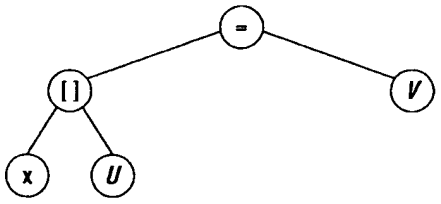
*Primary Examiner*—Kakali Chaki
*Assistant Examiner*—Hoang-Vu Antony Nguyen-Ba

(57) **ABSTRACT**

A semantic-based system is provided to upgrade software written in a high level language of the kind having a type system and being statically compiled in compilers which check types and usages at compile time. The system employs conversion declarations which are inserted into the source code of the software and executed by the compiler. These declarations comprise a list of substitutable parts, with each substitutable part having a list of properties, a set of semantic patterns to be matched using the substitutable parts, and a result pattern showing what will be substituted for each matching portion of source code.

**16 Claims, 3 Drawing Sheets**




NORMAL OPERATION

| SIZE | OPERATION | ... 1st OPERAND REP... | ... 2nd OPERAND REP... | ... |
|---|---|---|---|---|

VARIADIC OPERATION

| SIZE | OPERATION | COUNT | ... 1st OPERAND REP... | ... 2nd OPERAND REP... | ... |
|---|---|---|---|---|---|

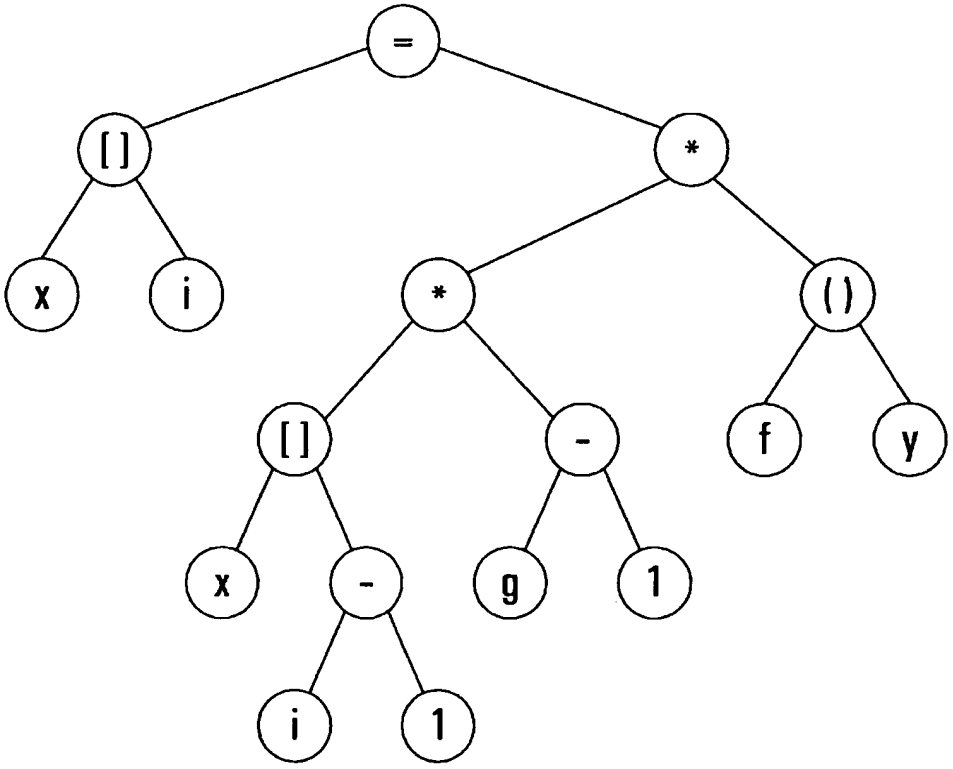



FIG. 1

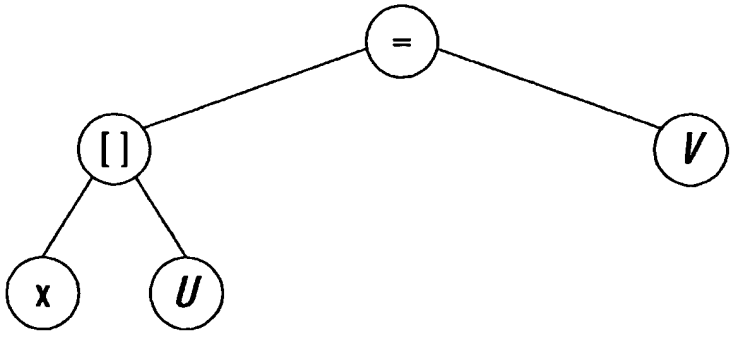



FIG. 2

NORMAL OPERATION

| *SIZE* | OPERATION | ... 1st OPERAND REP... | ... 2nd OPERAND REP... | ... |
|---|---|---|---|---|

VARIADIC OPERATION

| *SIZE* | OPERATION | COUNT | ... 1st OPERAND REP... | ... 2nd OPERAND REP... | ... |
|---|---|---|---|---|---|

FIG. 3

| *SIZE* | FOR | *... INIT-EXP REP...* | *... TEST-EXP REP...* | *... STEP-EXP REP...* | *... BODY-STMT REP...* |
|---|---|---|---|---|---|

FIG. 4A

| *SIZE* | BLOCK | 23 | *... 1st STMT REP...* | *... 2nd STMT REP...* | *... 3rd STMT REP...* | ... |
|---|---|---|---|---|---|---|

FIG. 4B

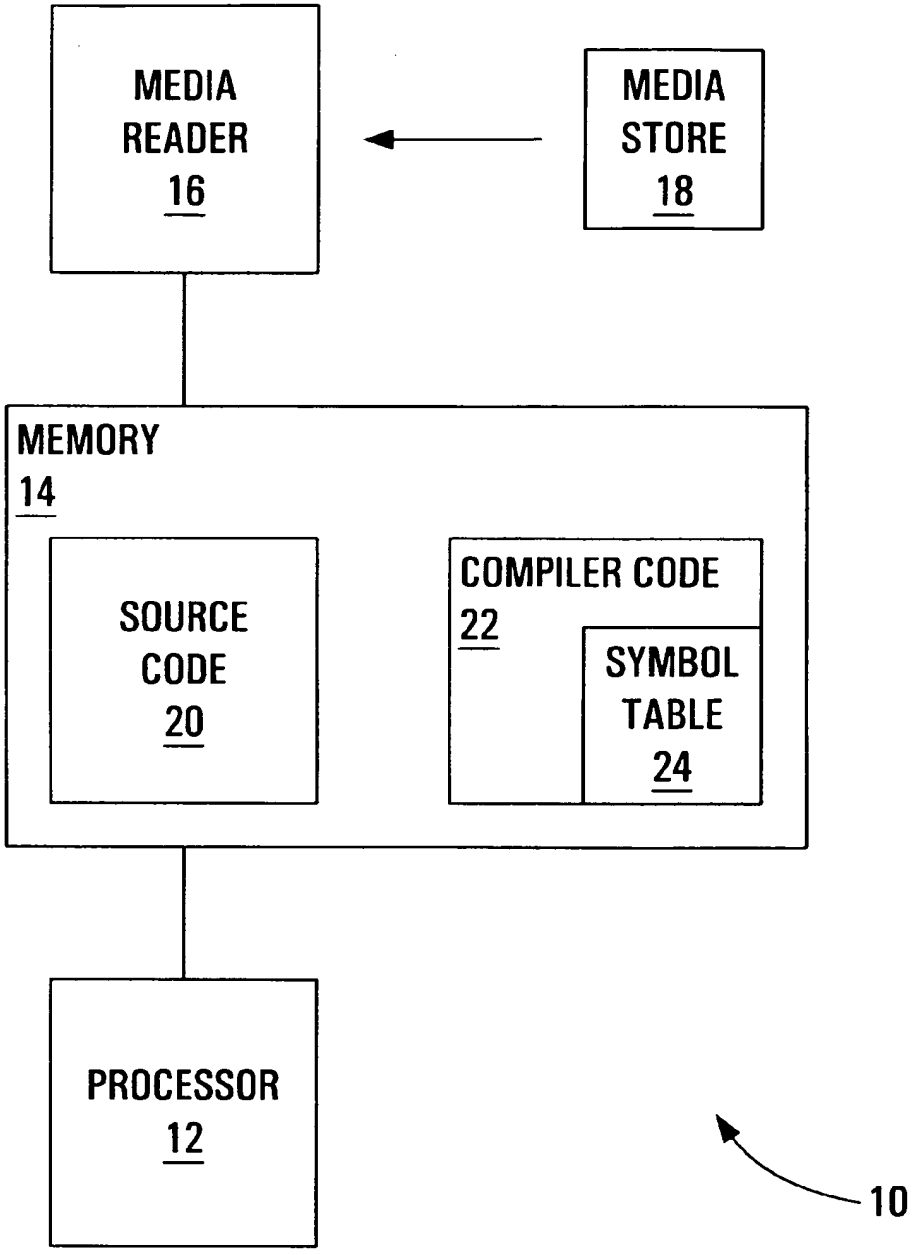

MEDIA READER 16
MEDIA STORE 18
MEMORY 14
SOURCE CODE 20
COMPILER CODE 22
SYMBOL TABLE 24
PROCESSOR 12
10

FIG. 5

# SOFTWARE UPGRADES BY CONVERSION AUTOMATION

## BACKGROUND OF THE INVENTION

This invention relates to a conversion declaration for use in upgrading source code of a program and to a programming compiler for implementing conversion declarations.

Most high level programming languages used in writing large software programs (e.g., JAVA, C, C++, COBOL, FORTRAN-IV, FORTRAN-90, ADA, PL/1, MODULA-2, MODULA-3, EIFFEL, ML, HASKELL, SATHER) have a type system and are statically compiled in compilers which check types and usages (i.e. semantic usages) at compile time. A "type system" means that each variable in the program has a type associated with it. Example types are integers, floating point numbers, and characters. It is often necessary to upgrade such a software program. To facilitate this process, text-based pattern matching and substitution systems are known. However, such systems have a number of drawbacks which may be highlighted by considering the following C code fragment:

```
01  { int my_var = 7;
02
03  struct MS {
04        float my_var;
05        char  code;
06  } my_struct;
07
08  my_struct.
09        my_var = 12.0 + my_struct.my_var;
10
11  {     char my_var = 'M';
12        char other_var = my_var + trunc(my_struct.my_var);
13        struct MS *my_var_ptr = &my_struct;
14        if    (my_var_ptr->my_var) printf(stdout,   "non-
          zero ");
```

A text-based pattern matching and substitution system cannot distinguish among the three different variables named my_var, declared on lines 01, 04, and 11, without heroic efforts. That is because the semantics of an identifier can be highly diverse within a single program in almost any high-level programming language.

A text-based system also has considerable difficulty distinguishing between the two different kinds of accesses to my_struct.my_var on line 09: the first occurrence is a write access, and the second is a read access.

A text-based system cannot tell that the expressions my_struct.my_var on line 09 and my_var_ptr->my_var on line 14 are denotationally identical (except possibly for the structure instance affected), because they do not 'look' alike.

Similarly, a text-based system cannot easily distinguish among expressions by type: for example, it cannot readily recognize that on line 12, my_var is a char-valued expression (that is, it is a character type expression), whereas trunc (my_struct.my_var) is an int-valued expression (that is, it is an integer valued expression).

Another problem area for text-based pattern matching and substitution is that, due to the lack of type- and usage-checking associated with text-based substitutions, the changes made in the program text must be checked by responsible, expert programmers. This creates two difficulties:

1. Checking is significant work for the responsible expert programmers, especially where the program is large and the changes are not isolated to a small portion of the program.



2. The text delta (list of changes in the program text) may be very large, even where (semantically speaking) what is happening is simply a specific change which could be stated, in English, in a sentence or two. (For example: "Change all read references to data_base_index to current_env.data_base_index. Change all write references to data_base_index from data_base_index=Newvalue to log_update (data_base_index,NewValue.")

This makes tracking the system changes a huge job: it hides individually significant changes by burying them in masses of stereotyped changes.

A final problem area for text-based pattern-matching and substitution systems is that they are not scoped in the same way as normal program entities. Consider, for example, the following C code fragment:

```
01        extern int K;
02        #define incr(X)   (X += K)
                . . .
03        void next_step (double J,K) {
04        {
05            int count = current_count;
06            incr(count);
```

(The " . . . " represents omitted material.)

If a programmer wrote code along the above lines, then the effect of the call to the incr macro on line 06 would almost certainly not be what the programmer intended. Its effect, as written, would be to increase the value of the local count variable by the value of the double parameter, K. However, from the declaration context on lines 01–02, it appears that the intent would be that count should be incremented by the value of the extern int K.

The problem is that the scope of macros is entirely different from that of ordinary declared program entities (variables, constants, routines, types, etc.). Here, the handling of incr is done by the C pre-processor (cpp) which uses a completely different processing strategy from the C compiler proper, and takes no cognizance of C scoping rules.

This invention seeks to overcome drawbacks of prior software upgrading systems.

## SUMMARY OF INVENTION

According to the present invention, there is provided a conversion declaration for use in upgrading source code of a program which has a type system and is statically compiled in a compiler which checks types and usages at compile time, comprising: a list of substitutable parts, with each substitutable part having a list of properties; a set of one or more semantic patterns to be matched using said substitutable parts; and a result pattern showing what will be substituted for each matching portion of said source code.

A compiler utilizing such a conversion declaration is also provided.

According to another aspect of the present invention, there is provided a method for upgrading a source code of a program of a type which has a type system and is statically compiled in a compiler which checks types and usages at compile time, comprising the steps of: comparing portions of said source code with semantic patterns in a set of one or more semantic patterns and a list of substitutable parts, with each substitutable part having a list of properties; on finding a matching source code portion, determining whether said matching source code portion comprises a pre-selected type, procedure or module and, if so, converting said matching source code portion to a result pattern.

## BRIEF DESCRIPTION OF THE DRAWINGS

In the figures which illustrate preferred embodiments of the invention,

FIG. 1 is a representation of a semantic tree,

FIG. 2 is a representation of a semantic pattern,

FIG. 3 is a Polish form representation of a source code fragment,

FIGS. 4*a* and 4*b* are Polish form representations of source code fragments, and

FIG. 5 is a schematic representation of a computer system embodying the subject invention.

## DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

The subject invention employs a semantic-based conversion of a software program in order to effect a modification to the program. This conversion is accomplished by the use of conversion declarations (together with any necessary declarations which define variables used in the conversion declarations) which are inserted at appropriate places into the source code and by appropriate modification to the compiler in order that it may handle the conversion declarations. At compile time, these conversion declarations are executed, as many times as necessary, to implement the desired modifications throughout the program.

To facilitate understanding of conversion declarations, two known semantic devices are first explained: semantic trees and semantic patterns.

Semantic Trees

Consider the C expression:

$$x[i]=x[i-1]*(g-1)*f(y)$$

where the "=", following C rules, denotes assignment of its right operand to its left operand (and not comparison or equality).

This expression is represented as the semantic tree of FIG. 1. It will be noted that the form of the tree follows from the precedence and associativity of the operators and the parenthesizing of the operands.

The leaf nodes (x, i, x, i, 1, g, 1, f, y) represent operands. The internal nodes represent language-defined operations which, in this case are "=" meaning "assign"; "[]" meaning "index"; "*" meaning "multiply"; "−" meaning "subtract; and "()" meaning "function call".

Note that textual details such as parentheses (where they do not denote the function call operation) are omitted. Grouping of operands is implied by the semantic tree with no need to include the parentheses as one might for a text-oriented parse tree. Ordering of the operand edges in the tree from left to right is significant, because x-y and y-x are quite different semantically. Even sub-trees composed entirely of mathematically associative and commutative operations, such as addition, cannot actually be restructured without a change of semantics when hardware floating point operations are used for implementation. Properties, such as loss of significance, can depend greatly on expression tree structure, so what the programmer coded should be what happens.

FIG. 1 illustrates all but one essential property of a semantic tree: All mode identifications denoting specific entities or operations must be exact. For example, the type of the operands is relevant to the operation performed for all interior nodes, and the exact denotation of variables and functions also. Thus, the node identification for a node labelled "u" would distinguish the type of u and distinguish that particular u from any other distinctly declared u in the program.

For example, if u were a long int (i.e., a long integer) declared in the second block nested within the third block nested within function h, then the identification for a node labelled u might actually be something like:

```
( name:          u
  type:          long int
  scope:         local
  function:      h
  block_loc:     (3,2)
)
```

whereas if u were a float (i.e., a floating point number) declared to be global in scope as an extern, then its identification might actually be something like:

```
( name:          u
  type:          float
  scope:         extern
  function:      ~
  block_loc:     ~
)
```

The use of "~" does not mean "unspecified". It means "absent" or "not applicable here". That is, use of "~" is precise. It indicates something which is definitely not present or applicable.

Type information for such specifications must be complete. For example, a bit-field three bits wide might have a type specified by

type: unsigned int :3

and an array parameter in a function might have a type specified by

type: float [50] []

That is, bit-field width and known array dimension(s) is relevant "type" information.

Individual occurrences of an identified identity such as the two different kinds of u above also need access mode information. Which access modes exist will depend on the programming language. For example, a variable in C can be accessed in the following ways:

1. Variable address (var_addr): An example is the C expression, &x which takes the address of an addressable variable x. This includes the case of a structure or union where a member is addressed.

2. Constant address (const_addr): Same as 1., except that the variable addressed has the const attribute. This includes the case of a constant structure or union where a member is addressed.

3. Assign to addressable variable (var_assign): The mode of an addressable non-const variable which is the left operand of = in an assignment or the initialization of a declaration.

4. Read and assign addressable variable (var_update): The mode of an addressable, non-const variable which is the operand of prefix or suffix ++ or — or the left operand of +=, −=, *=, /=, %=, &=, =, |=, <<=, or >>=, or the mode of a struct or union variable when one of its members is modified in part, or read and assigned, or simply overwritten. The reason that a union member assignment is not treated as having var_assign access mode is because there may be 'slack bits' not occupied by the assigned member, members of a union need not all have the same size in memory.

5. Initialize addressable constant (const_assign): The access mode of a newly declared non-bit-field const variable which is being initialized by = in a declaration.

6. Read addressable variable (var_read): the access mode of any addressable non-const variable in all contexts except those mentioned in 1., 3., and 4., including the access mode of a structure or union when a member is read-accessed.

7. Read addressable constant (const_read): the access mode of any addressable const variable in all contexts except those mentioned in 2. and 5., including the access mode of a constant structure or union when a member is read-accessed.

8. Assign to bit-field (bit_assign): the access mode of any bit-field variable which is the left operand of = in an assignment.

9. Read and assign bit-field variable (bit_update): The mode of a bit-field variable which is the operand of prefix or suffix ++ or -- or the left operand of one of +=, -=, *=, /=, %=, &=, =, |=, <<=, or >>=.

10. Read bit-field (bit_read): the access mode of any bit-field variable in all contexts except those mentioned in 8. and 9.

11. Pure value (pure_val): the access mode of a pure value; i.e., one not immediately tied to any memory location, such as the value of an expression such as "(x+1)" or the value returned from a function call.

12. Unvalued (no_val): the access mode of a statement, or of an expression preceding a "," operator—whether or not it provides a value, it certainly provides no value which is used.

Note that fine distinctions are made among modes in the C language. The access mode has a profound effect on what semantic substitutions are legitimate (for example, for mode 9., a substitution which used the & (pointer to) operator, which needs an addressable operand, would not be appropriate).

Thus, a full identification of the foregoing node labelled "u" would be as follows:

| | |
|---|---|
| ( name: | u |
| type: | float |
| scope: | extern |
| function: | ~ |
| block_loc: | ~ |
| access_mode: | var_read |
| ) | |

where the access mode specifies that u is a read-only variable.

Semantic Patterns

A semantic pattern is a semantic tree in which the specification of one or more nodes may have been made less precise in order to allow various semantic trees to match it. In fact, semantic trees comprise that subset of semantic patterns in which all nodes are precisely identified.

An example semantic pattern is illustrated in FIG. 2. Referencing FIG. 2, capitals are used for the names of pattern wild-cards to distinguish them from ordinary declared variable and constant names.

If, referencing FIG. 2, x is an array of type float and U and V are specified as follows:

| | |
|---|---|
| ( name: | U |
| type: | long int |
| scope: | ? |
| function: | ? |
| block_loc: | ? |
| access_mode: | var_read \| var_assign |
| ) | |

(where the "var_read|var_assign" access mode specification indicates that a match is possible for either of var_read or var_assign access mode, and "?" is used to indicate "don't care", i.e., anything matches this part of the pattern element specification)

| | |
|---|---|
| ( name: | V |
| type: | float |
| scope: | ? |
| function: | ? |
| block_loc: | ? |
| access_mode: | var_read |
| ) | |

then the semantic pattern of FIG. 2 matches the semantic tree of FIG. 1.

Note that a pattern variable differs from an ordinary declared variable or constant in being incompletely specified, either through the use of alternatives (as in "var_read|var_assign" above) or through the use of 'don't cares' specified above by "?".

Conversion Declarations

The essential information which is provided by a conversion declaration is as follows:

1. A list of substitutable parts and their properties (with information on what parts of their descriptions are fixed, which have multiple options, and which are "don't cares" as in the description of U and V above).

2. A set of semantic patterns to be matched using the substitutable parts provided in 1., including precise specifications of the fixed parts of the semantic patterns. In this regard, the context of the conversion declaration can typically be used to set some aspects of the fixed parts of the semantic patterns. For example, where a fixed part is represented by a name, say, "xyz", that name will have all of the properties which are tied to it at the point in the program at which the conversion declaration occurs. By way of explanation, any name has a "scope" in the program, which is the parts of the program in which the name is visible. For example, the name "xyz" could be declared in two blocks, one nested within the other. In each block, the name could have entirely different properties. If a conversion declaration were added to the outer block and used "xyz" as a name representing the source pattern, "xyz" would have the properties assigned to it in the outer block.

3. A result pattern which shows what will be substituted for the matching source program entity. A result pattern may also have fixed parts, if so these follow the same scoping rules as those followed in respect of source patterns.

4. What the conversion is tied to, i.e., the conversion can be tied to a type, a procedure (e.g. sub-routine), or a module. If the conversion is tied to a type, an otherwise matching source fragment will only trigger the conversion if the source fragment type matches the type specified in the conversion declaration. On the other hand, if the conversion is tied to a variable or procedure, then an otherwise matching source fragment will only trigger the conversion if the

source fragment matches the variable or procedure specified in the conversion declaration. The conversion may also be tied to some entity other than a type, such as use in a particular module. In the case of C, the issue of modules does not arise. After the C pre-processor, cpp, has been run, the compiler is dealing with just one single source program contained in one text file.

A conversion declaration is scoped in the same fashion as routines and types, with symbols in them fixing their meanings at the point where the declaration occurs. With a conversion declaration, then, the meaning of an expression such as "extern int K" is fixed at its declaration, and any later, more local, definition of K has no effect on the conversion declaration. Thus, conversion declarations are lexically scoped.

This essential information may be specified in a conversion declaration in many different ways. In a preferred form, a conversion declaration has the following syntax: convert entire_mode [self_mode; $parm_1$, $parm_2$, . . . , $parm_m$] $source\_pattern_1$; $source\_pattern_2$; . . . ; $source\_pattern_n$; to result_pattern;

The parts above are to be understood as follows:

$parm_1$, $parm_2$, . . . , $parm_m$

The parameters specify (i.e., list the properties of) the substitutable parts in the semantic patterns generated by the conversion declaration. In some conversion declarations, there will be no parameters.

In the preferred embodiment, the syntax for any one parameter appearing in the

$parm_1$, $parm_2$, . . . , $parm_m$ part is:

access_mode_alternatives type name

where access_mode_alternatives either have the form of a list of one or more access modes separated by "|":

$access\_mode_1|access\_mode_2| \ldots |\ access\_mode_k$

or are specified as

any_mode

(meaning any access mode is an acceptable match, in other words, the access_mode is a "don't care" attribute), and each specifically indicated access_mode is one of:

| | | |
|---|---|---|
| var_read | const_read | bit_read |
| var_assign | const_assign | bit_assign |
| var_update | const_addr | bit_update |
| var_addr | | |
| no_val | | |

The name is the name of the substitutable part.

$source\_pattern_1$; $source\_pattern_2$; . . . ; $source\_pattern_n$;

These are source language fragments, such as expressions or statements, which together with the information provided by the parameters, and the information provided by the properties of names in the context in which the conversion declaration appears, are sufficient to generate the semantic patterns to be matched when the conversion declaration is applied.

result_pattern

The result_pattern determines the result of applying the conversion declaration. Suppose a match occurs between a particular source fragment $source\_pattern_i$ (viewed as a semantic tree: the matching semantic tree) and the semantic pattern produced for $source\_pattern_i$. Then the result is found by converting the semantic pattern produced by the result_pattern into a semantic tree by substituting for the substitutable parts in the result_pattern those subtrees of the matching semantic tree which match them when $source\_pattern_i$ is matched to the matching semantic tree. Since the results are precise, the substitutions turn the result_pattern in to the result semantic tree. The compiler generates code as if the result semantic tree, rather than the matching semantic tree, had been written in the first place.

entire_mode

This specifies the permitted access mode(s) of the entire matching source fragment. A match will not occur unless the matching program entity has one of the specified access modes.

self_mode;

"self" is a reference to the type or class in which the conversion declaration is embedded. "self_mode" is used to restrict what access modes for instances of "self" are considered matches.

The $source\_pattern_1$; $source\_pattern_2$; . . . ; $source\_pattern_n$; portion of the conversion declaration may sometimes use patterns containing multiple statements. To handle such cases in C or C ++, multiple statements may be grouped with "[ . . . ]" brackets within the source_patterns, since "[" can never begin a statement in C or C ++. The same device can be used where such groupings are required in the result_pattern. Other techniques can be used, depending on the programming language.

The preferred syntax for a conversion declaration may be better understood by way of the following examples in the C language.

Consider first a simple example involving a system which handles transactions. To keep track of the number of transactions processed, the system increments a transactions completed count, maintained as a long integer:

++trans_count—or—trans_count++

The system is then modified to process transactions in parallel. The former code for updating trans_count is no longer safe, because the increment is not guaranteed to be atomic. Therefore all such updates need to be replaced with one of the following calls, which call routines implemented to accomplish the equivalent increment actions atomically:

long pre_incr_trans_count () /*if the value is used and was pre-incremented.*/

long post_incr_trans_count () /*if the value is used and was post-incremented.*/

incr_trans_count () /*if the value is not used and was pre-incremented or post-incremented. */

All of this can be accomplished system-wide by the following three conversion declarations (included in an appropriate C header file):

convert pure_val [] ++trans_count; to pre_incr_trans_count();

convert pure_val [] trans_count++; to post_incr_trans_count();

convert no_val [] ++trans_count; trans_count++; to incr_trans_count();

The pure_val and no_val modes above constitute the entire_mode specification, indicating that the expression to be substituted (++trans_count or trans_count++) in its entirety, has the pure_val access mode, since the value of either ++trans_count or trans_count++ is the value of (trans_count+1), which has the pure_val access mode. (The access mode of trans_count itself in the above is, of course, var_update.)

Note that the above conversion declarations would have to be declared where the system-wide trans_count was already defined. The implementation would use the declared meaning of trans_count in its semantic patterns for the above conversion declarations, so that if any programmer declared some other entity named trans_count, whether of the same or a different type, it would not match the semantic

patterns based on the above declared meaning of the trans_count name, and would not in any way be affected by the above three conversion declarations.

Now consider a more complicated example involving a (serial) transaction procession system. The bodies of software handling different kinds of transactions are distinct, and are associated with unsigned short identifier values, so that body of software which is handling a transaction can be tracked during execution.

Each transaction is associated with a struct called a trans_context which keeps track of data associated with that transaction. Included in a trans_context is a trans_owner member, which indicates the body of software which originally was selected to handle the transaction and continues to have overall responsibility for handling it. Different trans_contexts can be linked together during execution so that one transaction's handling can be modified depending on the data of other transactions which are in progress.

When one transaction-handling body of software 'visits' a transaction by a link, rather than directly accessing its owned transaction, it records that fact by setting the visited transaction's last_visitor member, so that, if something goes wrong, dependencies on linked transactions can be tracked for debugging purposes.

Suppose a modification to this system is being considered which will convert it from serial to parallel form, where each transaction-handling body of software runs as a sequential process, but the various transaction-handling bodies of software run in parallel with one another. However, before doing so, an estimate is required of the mutual-exclusion overhead which will be incurred by visits to linked transactions. Sometimes a linked transaction will turn out to be simply the current owned transaction, and would therefore incur no added overhead. Therefore, we need to count those visits in which the visited transaction is not the current transaction before any changes are made to make the system parallel.

Suppose x is some trans_context, and we are visiting it by a link (a pointer in some other context's data). We indicate that we are visiting it by executing

x.last_visitor=my_id;

where my_id denotes whatever declared or literal constant or variable holds the identifier for the transaction-handling body of software in which this code is executed.

A fragment of the data structures in this system is shown, as it appears after addition of conversion declarations to perform the needed counting of visits to unowned transactions:

```
01  extern unsigned long non_owner_visit_count; /*ADDED*/
02  struct trans_context {
03    unsigned  short       trans_owner;
04    struct    trans_data  *input,   *output;
05    unsigned  short       last_visitor;
        . . .
06  convert no_val                                  /*ADDED*/
07      [  var_update;
08         var_read|const_read|pure_val
09                unsigned short     visitor_act_id  ]
10            self.last_visitor = visitor_act_id;
11      to
12            [  self.last_visitor = visitor_act_id;
13               if (visitor_act_id != self.owner)
14                  ++non_owner_visit_count;
15            ];
16  } /*end struct trans_context*/;
```

What has been added to the original system to perform the counting is shown above as the two declarations marked by the "/*ADDED*/" comments: namely, the declaration of the non_owner_visit_count and the conversion declaration within the trans_context struct's declaration. The only other things left to do to complete the counting job is to add code to place the non_owner_visit_count in global memory with an initial value of zero, and to output the contents of the non_owner_visit_count at an appropriate point, and then run the system. All the rest of the work is done by the two added declarations.

Let's look at the details of what the above conversion declaration actually says. First, the conversion declaration (lines 06–15) is placed within the trans_context struct (lines 02–16). This indicates that the conversion is triggered only in the presence of an instance of the type "struct trans_context" (specifically, in the program context shown, by the appearance of self in the source_pattern on line 10).

no_val (line 06) is an entire_mode which indicates that we are interested only in cases where the entire affected code fragment does not return a value (i.e., appears as a statement). The self_mode, var_update, (line 07) indicates that we are looking for cases where the instance of the type "struct trans_context" is being updated. The next two lines define properties of a substitutable part, visitor_act_id, for the semantic patterns. Specifically, var_read|const_read|pure_val indicate that visitor_act_id may be a readable variable, readable constant, or some other value-oriented expression. unsigned short indicates the visitor_act_id is of type unsigned short.

The source _pattern (line 10) says that we are looking for the occurrence of an assignment in which the last_visitor member of an instance of the type "struct trans_context" is assigned the value of our substitutable part (i.e., last_visitor is assigned an unsigned variable, constant or pure value). The appearance of self represents where in the source pattern our instance of the type "struct trans_context" must appear.

The result_pattern (lines 12–15) says that, when a match occurs on the above criteria, we replace the matched fragment (an assignment statement) by the statements within square brackets, namely the same assignment statement followed by an if-statement which checks to see whether the value of the substitutable element, visitor_act-id, is the same as the owner member in the matching instance of type "struct trans_context", and if not, increments the non_owner_visit_count variable.

For conversion declarations to operate properly in all situations, two additional capabilities are required—avoiding recursion and handling deletions—which are discussed here following.

Literal Source Fragments

Since conversion declarations work by substitution prior to object code generation, rather than by closed linkages, they cannot invoke one another recursively. (To do so would be to incur an infinite expansion.)

However, there are occasions where we want the result_pattern of a conversion to be similar to a source_pattern. To prevent recursion in such cases, we need to provide a literal facility, which indicates that what the programmer wrote is to be taken as written, and not modified by any conversion declaration invocations. The syntax used for expressing this will of course, vary with the programming language in question. For C or C ++, the preferred embodiment is to allow source code forms such as any of:

literal selection_statement
literal iteration_statement
literal compound_statement
literal (expression)
literal [statement_list]

The meaning of a literal code fragment is its 'literal' meaning; i.e., its original meaning unmodified by any conversion declarations. The effect of making it literal is that neither the literal fragment, nor anything within it, can invoke a conversion declaration. In the preferred embodiment, the meaning of the statement, expression, or statement_list modified by literal is the same as it would be without the modifier literal, except for one thing: neither the entire literal-modified entity, nor anything within it, matches any source_pattern within a conversion declaration.

literal cannot be used within a source_pattern of a conversion declaration (source_patterns are implicitly literal to start with). It may, however, be used in a result_pattern, or in ordinary program source text.

As an example to motivate the use of a literal facility, consider the following problem: we want to find out how often entities of type struct data_handle are accessed in the execution of a system. This can be implemented as follows:

```
01 extern unsigned long data_handle_access_count;      /*ADDED*/
02 data_handle *access_data_handle (data_handle *);   /*ADDED*/
       . . .
03 struct data_handle {
       . . .
04 convert any_mode [any_mode;]                        /*ADDED*/
05       self;
06     to
07       literal(*access_data_handle(&self));
08 };
       . . .
09 data_handle *access_data_handle (data_handle *dh) (/*ADDED*/
10     ++data_handle_access_count;
11     return literal(dh);
12 }
```

In our example, the declarations marked "/*ADDED*/" have been added to the original system to support counting the accesses to data_handles during execution. We have added access to a counter in which to keep counts of accesses to data_handles (line 01). We also define a function to do the counting (line 02).

Inside the declaration of the struct data_handle type, we place a conversion declaration (lines 04–07), such that when we express a data_handle, x, in our code, it will be compiled as (*access_data_handle (&x)). Hence every execution of an access in the original program will increment the counter in the corresponding converted program.

Note the use of literal in the above. Without the use of literal, the result_pattern (line 07) would access itself at two points: the entire result_pattern, since it denotes a data_handle, would invoke the self-same conversion declaration, and self, denoting a conversion declaration, would also invoke it.

Absent Entities

Sometimes what is to be accomplished with a conversion declaration involves deleting a named entity (usually, replacing it with another). Yet conversion declarations use semantic patterns, which means that its patterns use names with semantics already attached. The named entity might be a global variable, a member in a structure, a routine, or any kind of entity with a name or type attached.

To support this, absent entities are provided. An absent entity has the same type and properties as its corresponding 'present' entity, except that it does not exist at run-time; i.e., it is a compile-time place-holder for type-checking and other semantic properties. How it is expressed will depend on the programming language in question.

When an entity is absent, then (aside from its declaration), it can only appear in conversion declaration source_patterns. Once all conversion declarations have done their work, all absent entities must have disappeared from the executable code.

In the preferred embodiment in C or C ++, absent entities are declared by using the word absent (indicating that entities of this type do not exist at run-time) as a type qualifier (like const, indicating that the entity cannot be modified, inline, indicating that a function is to generate in-line code where possible, or volatile, indicating that the variable may change at any time due to the action of multiple accessing threads of control).

For example, suppose we have a large system in which we have a date structure defined as:

```
struct date {
    unsigned int     dd:5,      /* Day of month */
                     mm:4,      /* Month of year */
                     yy:7;      /* Year minus 1900 */
};
```

where we have, assuming a suitable C implementation, packed the date into 16 bits, at the cost of limiting ourselves to years between 1900 and 2027, inclusive.

Suppose we discover that we must handle a wider range of dates: we now need years before 1900 and after 2027. However, the date structure has been used wholesale throughout our entire system. Conversion declarations together with an absent entity facility can address this problem, as follows.

We redeclare our date structure as follows:

```
01  struct date {
02    unsigned int     dd     : 5,     /* Day of month */
03                     mm     : 4;     /* Month of year */
04    signed int       yyyy   :23;     /* Year */
05    absent
06    unsigned int     yy     : 7;     /* Year - 1900 */
07    convert any_mode [var_update;
08                      var_read|const_read|pure_val int E]
09            self.yy = E;
10          to
11            self.yyyy = E + 1900;
12    convert var_read|const_read [var_read|const_read;]
13            self.yy;
14          to
15            self.yyyy - 1900;
16  };
```

The absent member declared on lines 05–06 takes up no space in a date and does not exist at run-time. Due to the conversions on lines 07–11 and 12–15, all read and write references to the old yy year bit-field are replaced by equivalent accesses to the new yyyy year bit-field, which has sufficient capacity to refer dates from 4, 194, 304 B.C. to 4, 194, 303 A.D. (assuming a suitable C implementation).

Equivalent Source Text Capabilities

So far, we have discussed basic uses of conversion declarations to implement wide-ranging changes in a large system using a few strategically placed conversion declarations, without further changes to source text.

However, there are cases where it may be desired to have access to equivalent source code changes as well. Some obvious examples are:

1. In a debugger, we may want to make the semantics of any invoked conversion declarations, in addition to what the programmer wrote, visible, so that the user can more easily understand what is happening in a debugging session.

2. Similarly, we may want to make visible the semantics of any conversion declaration invocations triggered by program fragments when an annotated compilation listing of the code is produced.

3. We may want to employ a particular conversion declaration as a temporary transitional step, planning to replace the source code in the long run so that the transitional conversion declaration is no longer needed.

For all of the above reasons, a compiler with conversion declaration capabilities may be deployed along with a source expander facility. The source expander can then be used by the debugger, the lister, and the library maintenance tools to address needs 1–3 above.

The source expander facility takes the source code and generates new source text which has exactly the same semantics as the original source text, but in which certain specific conversion declarations are no longer invoked. Generally, the source text changes required tend to be local to the invoking source fragments, but sometimes wider changes are required to preserve semantics.

Where conversion declarations are to be used as a transitional step in certain system-wide changes, it is recommended that conversion declarations be marked up so that conversion declarations which are transitional can be distinguished from those which are intended to remain. For example, we can begin transitional conversion declarations with something like, say,

pending convert . . . [ . . . —or—obsolete convert . . . [ . . .

where "pending" means that this conversion will eventually be eliminated by source transition to a new form, and "obsolete" means that the transition is imminent—the programmer should waste no time in applying the source expander to these transitions. Not only does this make the status of conversion declarations visible at a glance, but it allows the source expander to selectively transition invocations of marked conversion declarations, without requiring an explicit list of which conversion declarations are to be transitioned.

For example, consider the conversions to pre_incr_trans_count (), post_incr_trans_count (), and incr_trans_count () described previously. It might well be the case that we would want eventually to modify the system's source text so that the calls to these routines were made explicitly, instead of being implicit in the incrementation idiom and these three conversion declarations.

To make the transition, we would apply the source expander facility to the system's source text, identifying to it the three relevant conversion declarations. We can identify the conversion declarations by any combination of the following:

1. The source_patterns which appear in them; in this case ++trans_count or trans_count++ (or parts thereof).

2. The result_patterns which appear in them; in this case pre_incr_trans_count(), post_incr_trans_count(), and incr_trans_count() (or parts thereof).

3. A label name which we associate with them. To support this, the preferred syntax must be extended to permit a label, so that a conversion using a label begins with "convert CD_label_name: entire_mode [ . . . ".

A complication which arises in connection with use of an source expander facility is visibility or access control. In C, this is not an issue: opaque types are not much used. The only way to declare one is to use an incomplete structure, union, or enum type, declaration, such as "struct OPAQUE;", which declares a struct type named OPAQUE, with unknown members and unknown size. However, since conversion declarations require names, including member names, used in their patterns, to be defined, then any program fragment which can 'see' a conversion declaration which uses the member names must obviously have traversed the source text which declares them, and all code which 'sees' such declarations has exactly the same level of access to the declared entities.

In C++ and many other languages, however, visibility control can be a live issue. Generally, the problem is that certain members or components are permitted to be used only in certain scopes within the source text for the system. For example, private members of a C++ class can only be accessed by a member function or friend function or friend class of that class.

Note that the compiler still has complete information on such privileged components. Therefore, all that is needed to be able to facilitate the work of the source expander is to provide an easy way for the modified source text generated by the source expander to refer to elements with privileged access in the 'wrong' scopes.

In C++, the component access operators are "::", ".", and "->", used, respectively, to access class members, members in class instances, and members in target class instances accessed via a pointer. (Recall that struct and union types are considered special kinds of classes in C++.) To facilitate the work of the source expander, we need only provide 'privileged' versions of these operators—say, "::?", ".?", and "->?", respectively—and a compilation flag indicating whether or not such forms are permitted.

The issue here is that, being lexically scoped, conversion declarations declared within a C++ class, say, would have the same privileges as other code with the class, including privileged access to members. However, when we expand an invocation of a conversion declaration outside the class, this privileged access is revoked. We need the 'privileged access' operators to let the source expander produce a straightforward textual expansion with equivalent semantics. Of course, the source expander would only use the privileged access operators where they are actually required; all accesses which could be made without them would be expanded without them by the source expander.

Plainly, the privileged access operators should only be used to (1) make visible the semantics of the compiled code for debugging or listing purposes or (2) support short-term code patches. Uses beyond those would destroy the software engineering benefits of making certain accesses privileged.

Note that the textual expansions produced by the source expander are not equivalent to mere macro expansions or textually based substitutions. All of the aforenoted weaknesses of text-based systems are still avoided when the source expander is used selectively. Even wholesale use of the source expander avoids many of these weaknesses. The reason is that both the invocation mechanism and the substitution chosen remain semantically rather than textually based when the source expander is employed.

Implementing Conversion Declarations and a Source Expander

What follows is a discussion of how a compiler should be structured in order to support conversion declarations and a source expander facility, and what additional tools are needed to support the source expander capability.

It would be possible to have a compiler which converted its entire source program into internal linked data structures in the form of trees, together with a symbol table and type table. However, such an internal form for annotated program source would be excessively bulky.

We can, however, compactly represent the entire source of a large program as a flat string of bytes in Polish form, as shown in FIG. 3.

Polish form is a compact format for the representation of trees, where nodes generally represent some operation or pseudo-operation. The representation is simple: if a node has a fixed 'arity' (i.e., a fixed number of arguments or a fixed number of inputs), it is represented as a size, followed by the encoding of its operation or pseudo-operation, followed immediately by the representation of its operand subtree(s) (if any), from left to right. The size field makes it possible to navigate the tree swiftly. If we have to deal with nodes having many children, and we need more speed, we could 'index' the descendants to speed up access further. If it is variadic, it is represented as a size, followed by the encoding of its (pseudo-) operation, followed by an operand count, followed by the representation of its operand subtree(s) (if any). Each operand subtree has the same kind of representation; the leaf nodes are those with no further operand subtrees. The entire representation can easily be encoded as a flat sequence of bytes. The size is the number of bytes (or other storage units, if desired) needed to encode the subtree comprising the operation and its operands.

Even expanded to include line number information, such a representation can be about as compact as the original source. Unlike the original source, however, it can be designed to be almost as efficient to navigate as a normal pointer-linked data structure, especially if the above-mentioned 'indexing' of nodes with many children is used.

The representation of a for-statement, for example, would be as shown in FIG. *4a* where the init-exp rep is the representation of the initialization expression, the text-exp rep is the representation of the loop termination text expression, and the body-stmt rep is the representation of the statement which forms the body of the loop. The representation of a block with 23 statements in its body would be as shown in FIG. *4b*. (Note that for has a fixed count of four operands, whereas block is variadic.)

Turning to FIG. **5**, a computer system **10** comprises a processor **12**, a memory **14**, a computer media reader **16**, and a computer media store **18** (e.g., a computer diskette) storing compiler code. The memory stores a source program **20** and the compiler code **22** uploaded from disk **18**. The compiler code has a symbol table **24**. The processor **12**, under control of the compiler code **22**, acts as a compiler.

A suitable representation for conversion declarations in the compiler symbol table **24** uses:

1. A representation similar to that used for routines for the handling of their parameter and access mode information.

2. Polish form for source_patterns and result_patterns, with some special encoding for references to parameters such as beginning the occurrence of a parameter with an otherwise illegal character or character sequence, and representing the parameter as a numeric value stored as one byte.

Conversion declaration information can be placed in the compiler's symbol table as follows:

1. For each type declaring or tied to a conversion declaration, the type information contains an enumeration of the various conversions associated with that type. (The exact implementation used to store this information is irrelevant; it could be a list, table, hash table, etc.). Occurrences of the type trigger an examination of the source_patterns of the conversion declarations associated with the type to see which, if any, apply.

2. Where a conversion declaration is not tied to a specific type, it is generally tied to the use of a particular bound name (i.e., a name tied to certain properties) such as a global variable or procedure definition. In these cases, the name table contains an enumeration of the various conversions associated with that entry. (The exact implementation used to store this information is irrelevant; it could be a list, table, hash table, etc.). Appearance of such a bound name triggers an examination of the source_patterns of the conversion declarations associated with the bound name to see which, if any, apply.

3. Conversion declarations not fitting one of the above two categories are not very useful and can be disallowed.

Thus, to convert a source code program, the processor **12**, under program control of the compile code **22**, reads the source code program **20** and compares the type or name of a portion of source code with types and names in the symbol table **24**. When a match is found, the processor, for each conversion declaration associated with the matching type or name, compares the source code portion with a source semantic pattern along with properties of its substitutable parts. On a match in respect of a given conversion declaration, the processor utilises the result pattern from the conversion declaration in place of the source code portion in generating object code. (In some cases it may be possible that the result pattern of the conversion declaration is written in object code rather than source code so that the compiler merely inserts this object code in place of the source code portion when compiling the program.)

To implement an equivalent source text facility, the compiler should control the replacement of source text matching a conversion declaration with new source text embodying the new semantics specified by the conversion declaration. All source text is exposed to the compiler, including line and column number information, as is the source text for all source_patterns and result_patterns. The compiler can either emit a control file indicating how the original source text must be modified to convert it to the new source text equivalent to the semantics specified by the expanded conversion declaration invocations, or can perform the changes directly.

On the whole, the way to implement the source changes is straightforward, but there is a potential significant complication: because conversion declarations are lexically scoped, like routines, it is entirely possible that the meanings of bound symbols at the point where the expansion is required are not the same as the meanings of those bound symbols where the conversion declaration was declared.

When this situation arises, its occurrence is exposed to the compiler. The compiler must then direct the expansion so that, not only does the required expansion occur at the invocation site, but any needed subsidiary changes must be made to ensure that the semantics are preserved despite the change in scope from the point at which the result_pattern appears in the conversion declaration and the point at which the result_pattern is applied at the conversion declaration invocation.

Consider a simple example:

```
01  typedef unsigned char bool;
      . . .
02  unsigned long XOI; /*ADDED: eXtra Overhead Incurred */
      . . .
03  convert no_val [] /*ADDED*/
04      ++trans_count; trans count++;
05    to
06      trans_count += 1 + XOI;
07  void order_update (bool XOI) /* eXtract Order Information */{
```

-continued

```
08    ++trans_count;
09  if (XOI) /* Handle eXtraction of Order Info: */ {
        ...
```

In the above source code fragment, we have a transaction processing system. We keep track of transactions processed in a variable called trans_count. The time comes when we want to weight transactions which incur extra overhead more heavily. When this will occur, we keep the weight in variable XOI declared on line 02. We also write a conversion declaration (lines 03–06) which will modify simple increments of trans_count to add in the value of XOI.

The problem arises when we want to generate equivalent source for the routine order_update starting on line 07. It takes a Boolean parameter, also called XOI, in this case meaning: eXtract Order Information. Our expansion must correctly refer to the XOI declared on line 02, not the XOI declared on line 07.

The solution is that the source expansion facility must rename one of the conflicting XOIs. Since the one declared on line 07 is local to the declaration of order_update, the most sensible choice is to rename this XOI, and to expand the declaration of order_update as shown below:

```
07  void order_update (bool zXOI) /* eXtract Order Information */{
08    trans_count += 1 + XOI;
        ...
09    if (zXOI) /* Handle eXtraction of Order Info: */ {
        ...
```

As shown, references to the XOI declared in order_date become references to zXOI. The conflict has been removed, and line 08 now performs exactly the semantics which the invocation of the conversion on it would have achieved.

Modifications of the invention will be apparent to those skilled in the art and, therefore, the invention is defined in the claims.

What is claimed is:

1. A compiler which checks types and usages at compile time, comprising:

a reader and storer for reading source code of a program having a type system and for, on encountering a conversion declaration for use in upgrading said source code, said conversion declaration having

a type or bound name to which said conversion declaration is tied;

a list of substitutable parts, with each substitutable part having a list of properties;

a set of one or more semantic patterns including said substitutable parts for matching one or more portions of said source code which are tied to said type or bound name; and

a result pattern showing what will be substituted for each matching portion of said source code,

storing a representation of said conversion declaration.

2. The compiler of claim 1 including:

a comparator for comparing portions of said source code program tied to said type or bound name with said semantic patterns including said substitutable parts of said stored representation of a conversion declaration and, based on said comparison, selectively substituting said result pattern of said stored representation of a conversion declaration for said portions of said source code program.

3. The compiler of claim 2 wherein said bound name comprises a procedure or module to which said conversion declaration is tied.

4. The compiler of claim 3 wherein said list of properties for a substitutable part comprises fixed information for properties that are fixed, permissible options for properties that have options, and a "don't care" indicator for properties which may have any possible option.

5. The compiler of claim 4 wherein said list of properties for a given substitutable part comprises access modes for said given substitutable part.

6. The compiler of claim 5 wherein each source pattern comprises multiple statements.

7. The compiler of claim 6 wherein said result pattern comprises multiple statements.

8. The compiler of claim 5 including an indication of an access mode for said source pattern.

9. The compiler of claim 2 wherein said reader and storer is also for adding to properties of fixed parts of said semantic patterns based on placement of said conversion declaration in source code, whereby said conversion declaration is lexically scoped.

10. A method for upgrading source code of a program of a type which has a type system and is statically compiled in a compiler which checks types and usages at compile time, comprising:

reading source code of said program and for, after encountering a conversion declaration having:

a type, procedure, or module to which said conversion declaration is tied;

a list of substitutable parts, with each substitutable part having a list of properties;

a set of one or more semantic patterns having said substitutable parts for matching one or more portions of said source code tied to said type, procedure, or module; and

a result pattern showing what will be substituted for each matching portion of said source code,

comparing portions of said source code which are tied to said type, procedure, or module with semantic patterns in said set of one or more semantic patterns and with said list of substitutable parts;

on finding a matching source code portion converting said matching source code portion using said result pattern.

11. The method of claim 10 further comprising adding to properties of fixed parts of said semantic patterns based on placement of said conversion declaration in source code, whereby said conversion declaration is lexically scoped.

12. A compiler which checks types and usages at compile time, comprising:

means for reading source code of a program having a type system and at least one conversion declaration for use in upgrading said source code, said at least one conversion declaration having

a type or a bound name to which said conversion declaration is tied;

a list of substitutable parts, with each substitutable parts having a list of properties;

a set of one or more semantic patterns having said substitutable parts for matching one or more portions of said source code tied to said type or bound name; and

a result pattern showing what will be substituted for each matching portion of said source code,

and for placing conversion declaration information in a compiler symbol table in association with said type or said bound name to which said conversion declaration is tied; and

means for, on encountering a type or bound name in a source code portion, comparing said encountered type or name with types and names in said symbol table and, on finding a matching type or name, for each conversion declaration tied to said matching type or name, comparing said set of semantic patterns having said substitutable parts of said each conversion declaration with said source code portion and on a match in respect of a given conversion declaration, substituting said result pattern of said given conversion declaration.

13. The method of claim 12 wherein said means for reading and placing is further for adding to properties of fixed parts of said semantic patterns based on a placement of said conversion declaration in source code, whereby said at least one conversion declaration is lexically scoped.

14. A computer readable medium for compiling a source code program, the computer readable medium executable in a computer system, comprising:

means for reading source code of a program having a type system and at least one conversion declaration for use in upgrading said source code, said at least one conversion declaration having

a type or a bound name to which said conversion declaration is tied;

a list of substitutable parts, with each substitutable part having a list of properties;

a set of one or more semantic patterns having said substitutable parts for matching one or more portions of said source code tied to said type or bound name; and

a result pattern showing what will be substituted for each matching portion of said source code,

and for placing conversion declaration information in a compiler symbol table in association with said type or said bound name to which said conversion declaration is tied; and

means for, on encountering a type or name in a source code portion, comparing said encountered type or name with types and names in said symbol table and, on finding a matching type or name, for each conversion declaration tied to said matching type or name, comparing said set of semantic patterns having said substitutable parts of said each conversion declaration with said source code portion and on a match in respect of a given conversion declaration, substituting said result pattern of said given conversion declaration.

15. The method of claim 1 wherein said means for reading and placing is further for adding to properties of fixed parts of said semantic patterns based on a placement of said conversion declaration in source code, whereby said at least one conversion declaration is lexically scoped.

16. A system, comprising:

a source code program having a type system and having at least one conversion declaration having:

a type or a bound name to which said conversion declaration is tied;

a list of substitutable parts, with each substitutable part having a list of properties;

a set of one or more semantic patterns having said substitutable parts for matching one or more portions of said source code tied to said type or bound name; and

a result pattern showing what will be substituted for each matching portion of said source code,

a compiler which checks types and usages at compile time for reading source code of said program and for, after encountering a conversion declaration in said source code, comparing portions of said source code program with said substitutable parts and said semantic patterns of said encountered conversion declaration and, based on said comparison, selectively substituting said result pattern of said encountered conversion declaration for said portions of said source code program.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE

# CERTIFICATE OF CORRECTION

PATENT NO. : 6,195,792 B1 
DATED : February 27, 2001
INVENTOR(S) : Mark Andrew Turnbull and Harold Joseph Johnson

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

Title page,
Item [75] Inventors, replace "Mark Andrew Turnbull; Harold Joseph Johnson, both of Nepean (CA)" with -- Mark Andrew Turnbull, of Stittsville (CA); Harold Joseph Johnson, of Nepean (CA) --.

Column 18,
Line 17, replace "The compiler of claim 2 wherein said reader and storer" with -- The compiler of claim 1 wherein said reader and storer --.
Line 56, replace "a list fo substitutable parts, with each substitutable parts" with -- a list] of substitutable parts, with each substitutable part --.

Column 19,
Line 11, replace "The method of claim 12 wherein said means for" with -- The compiler of claim 12 wherein said means for --.
Line 14, replace "said conversion declaration in source code, whereby said at" with -- said at least one conversion declaration in source code, whereby said at --.

Column 20,
Line 9, replace "The method of claim 1 wherein said means for reading" with -- The medium of claim 14 wherein said means for reading --.
Line 11, replace "of said semantic patterns based on a placement of said" with -- of said semantic patterns based on a placement of said at least one --.

Signed and Sealed this

Twenty-fifth Day of December, 2001

*Attest:*

*Attesting Officer*

JAMES E. ROGAN
*Director of the United States Patent and Trademark Office*

US006748587B1

(12) **United States Patent**
**Santhanam et al.**

(10) **Patent No.: US 6,748,587 B1**
(45) **Date of Patent: Jun. 8, 2004**

(54) **PROGRAMMATIC ACCESS TO THE WIDEST MODE FLOATING-POINT ARITHMETIC SUPPORTED BY A PROCESSOR**

(75) Inventors: **Vatsa Santhanam**, Campbell, CA (US); **David Gross**, Campbell, CA (US); **John Kwan**, Foster City, CA (US)

(73) Assignee: **Hewlett-Packard Development Company, L.P.**, Houston, TX (US)

( * ) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 0 days.

(21) Appl. No.: **09/002,404**

(22) Filed: **Jan. 2, 1998**

(51) **Int. Cl.**[7] ..... **G06F 9/45**
(52) **U.S. Cl.** ..... **717/140**
(58) **Field of Search** ..... 712/222; 395/705, 395/701; 708/495–497; 717/140–143, 127, 131

(56) **References Cited**

U.S. PATENT DOCUMENTS

| | | | | |
|---|---|---|---|---|
| 5,481,686 | A * | 1/1996 | Dockser | 712/222 |
| 5,729,724 | A * | 3/1998 | Sharangpani et al. | 712/222 |
| 5,761,105 | A * | 6/1998 | Goddard et al. | 708/510 |
| 5,812,439 | A * | 9/1998 | Hansen | 708/497 |
| 5,859,998 | A * | 1/1999 | Lynch | 712/222 |
| 5,884,070 | A * | 3/1999 | Panwar | 712/222 |

OTHER PUBLICATIONS

White Paper, "Intel C/C++ and FRTRAN Compilers", Intel 1997, Retrieved from the Internet Dec. 11, 1997) : <http://developer.intel.com/design/perftool/icl24/icl24wht.html>.*
Morales, "An SBNR Floating–Point Convention", IEEE 1998, pp 6–10.*
Kang et al., "Fixed–Point C Compiler for TMS320C50 Digial Signal Processor", ICASSP–1997, IEEE, Apr. 1997, pp 707–710.*
Sung et al., "Fixed–Point C Language for Digital Signal Processing", Proceedings of ASILOMAR–29, IEEE, 1996, pp 816–820.*
Gal et al., "An Accurate Elementary Mathematical Library for the IEEE Floating–Point Standard", ACM Transactions on Mathematical Software, vol. 17, No. 1, Mar. 1991, pp26–45.*
Markstein et al., "Wide Format Floating–Point Math Libraries", ACM, 1991, pp 130–138.*
Yernaux et al., "A High–Speed 22–bit Floating–Point Digital Signal Processor", ISCAS'88, IEEE, 1988<pp65–68.*
"Assembler Instructions with C Expression Operands," (Oct. 5, 1997) located at website: www.cygnus.com/pubs/gnupro/2; and "Asm Labels" (Mar. 19, 1996) located at website: euch6h.chem.emory.edu/services/gcc/html/Asm—specification notes accompanying releases of the Gnu C compiler; Cygnus Solutions, Inc.
Intel C/C++ and FORTRAN Compilers; white paper—copyright 1997, Intel Corporation.

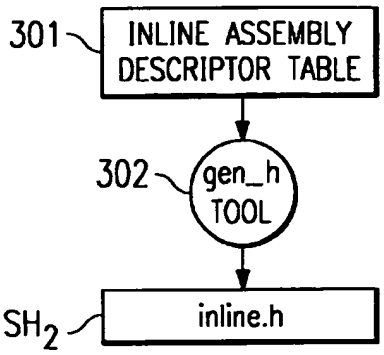
* cited by examiner

*Primary Examiner*—Wei Zhen

(57) **ABSTRACT**

A software mechanism for enabling a programmer to embed selected machine instructions into program source code in a convenient fashion, and optionally restricting the re-ordering of such instructions by the compiler without making any significant modifications to the compiler processing. Using a table-driven approach, the mechanism parses the embedded machine instruction constructs and verifies syntax and semantic correctness. The mechanism then translates the constructs into low-level compiler internal representations that may be integrated into other compiler code with minimal compiler changes. When also supported by a robust underlying inter-module optimization framework, library routines containing embedded machine instructions according to the present invention can be inlined into applications. When those applications invoke such library routines, the present invention enables the routines to be optimized more effectively, thereby improving run-time application performance. A mechanism is also disclosed using a "_fpreg" data type to enable floating-point arithmetic to be programmed from a source level where the programmer gains access to the full width of the floating-point register representation of the underlying processor.

**1 Claim, 3 Drawing Sheets**



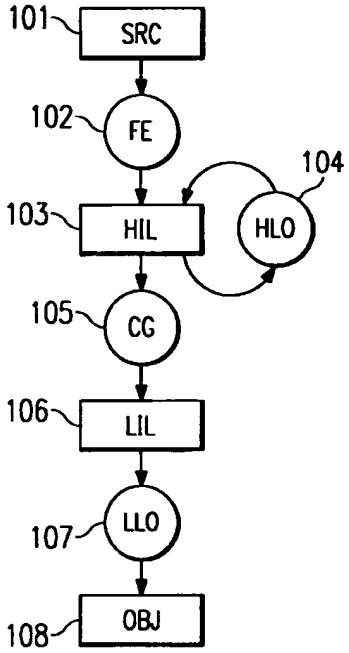



FIG. 1

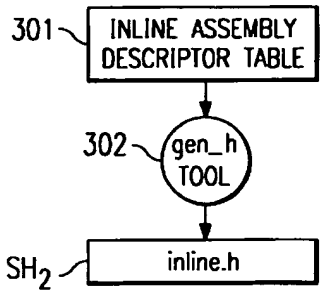



FIG. 3

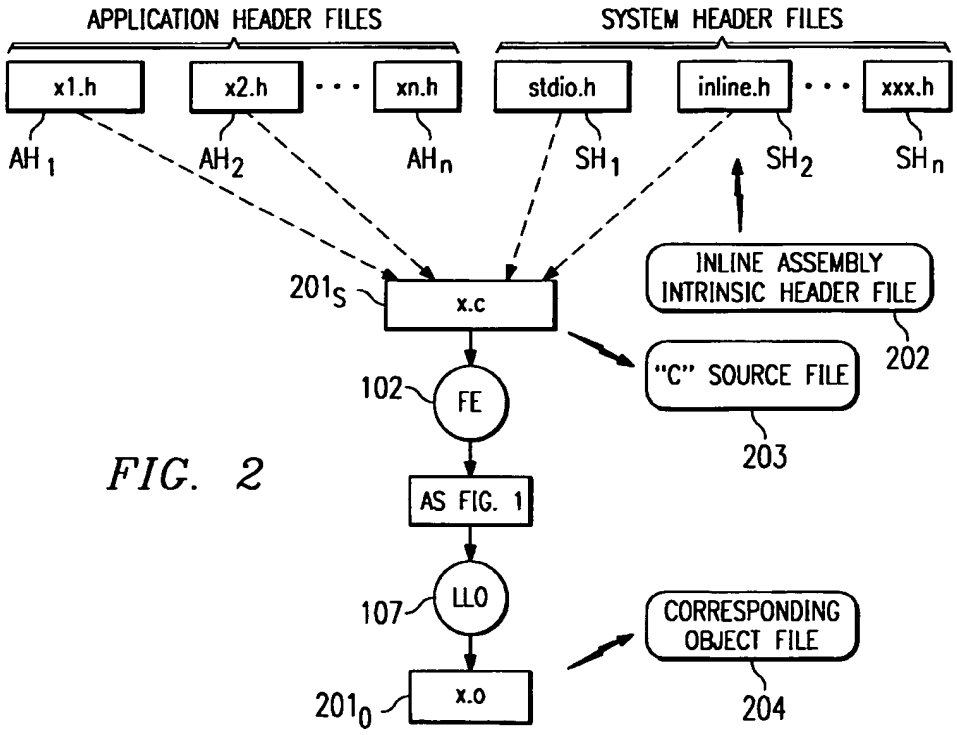



FIG. 2

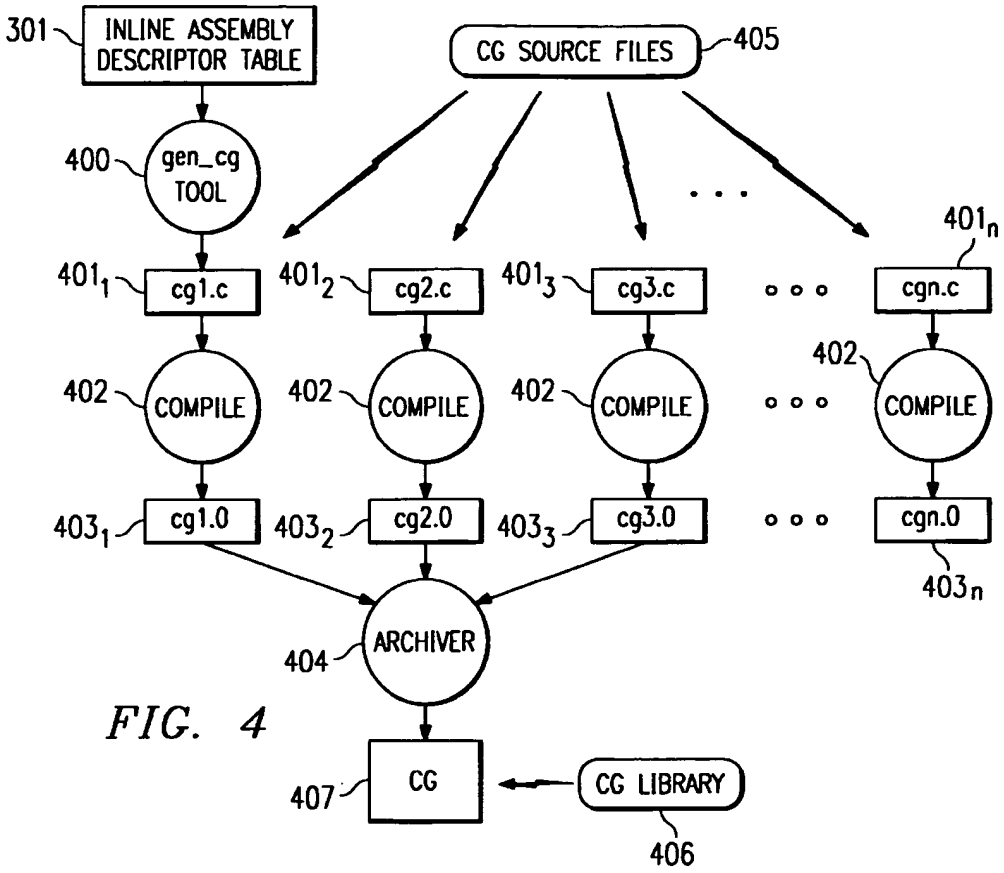

301
INLINE ASSEMBLY DESCRIPTOR TABLE
CG SOURCE FILES
405
400
gen_cg TOOL
401n
401 1
cg1.c
401 2
cg2.c
401 3
cg3.c
cgn.c
402
COMPILE
403 1
cg1.0
403 2
cg2.0
403 3
cg3.0
cgn.0
403n
ARCHIVER
404
CG
407
CG LIBRARY
406

FIG. 4

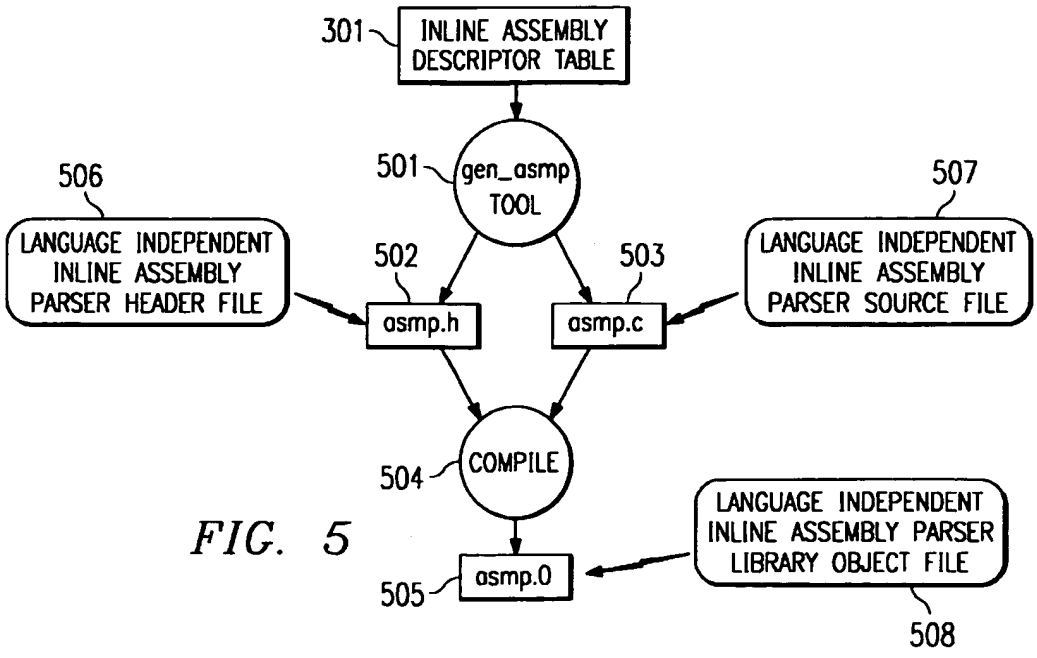

301
INLINE ASSEMBLY DESCRIPTOR TABLE
501
gen_asmp TOOL
506
LANGUAGE INDEPENDENT INLINE ASSEMBLY PARSER HEADER FILE
507
LANGUAGE INDEPENDENT INLINE ASSEMBLY PARSER SOURCE FILE
502
503
asmp.h
asmp.c
504
COMPILE
LANGUAGE INDEPENDENT INLINE ASSEMBLY PARSER LIBRARY OBJECT FILE
505
asmp.0
508

FIG. 5

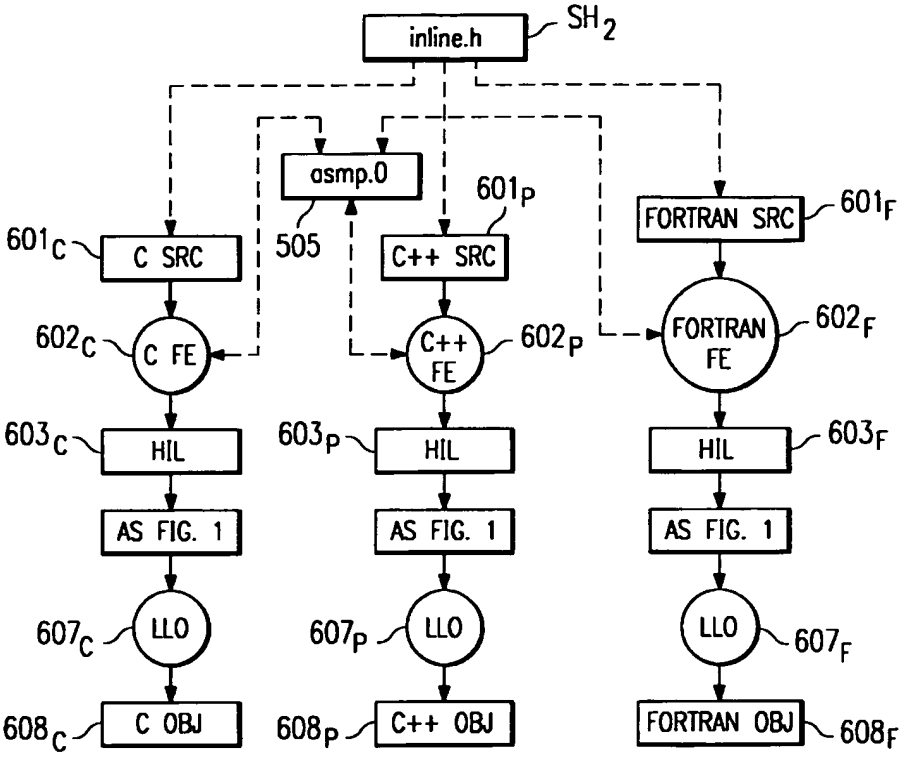

inline.h
SH2
asmp.0
505
601P
601C
C SRC
C++ SRC
FORTRAN SRC
601F
602C
C FE
C++ FE
602P
FORTRAN FE
602F
603C
HIL
603P
HIL
HIL
603F
AS FIG. 1
AS FIG. 1
AS FIG. 1
607C
LLO
607P
LLO
LLO
607F
608C
C OBJ
608P
C++ OBJ
FORTRAN OBJ
608F

*FIG. 6*

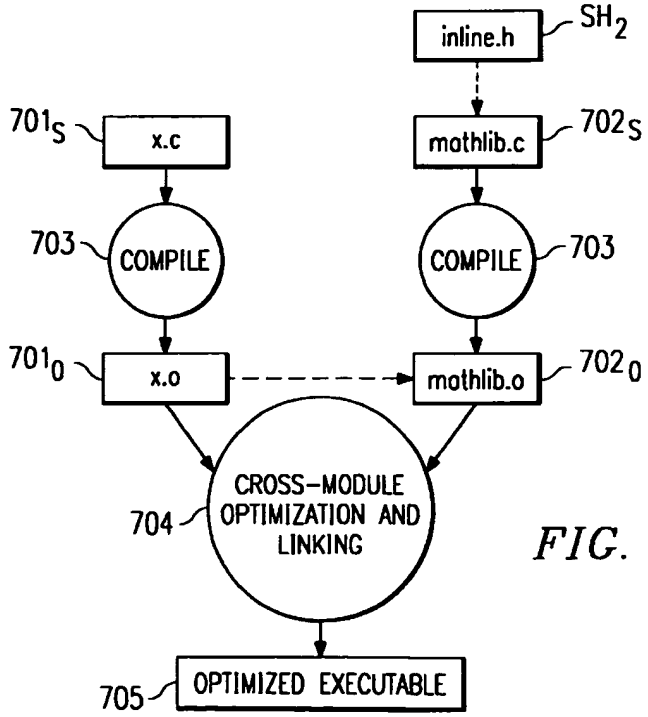

inline.h
SH2
701S
x.c
mathlib.c
702S
703
COMPILE
COMPILE
703
701O
x.o
mathlib.o
702O
704
CROSS-MODULE OPTIMIZATION AND LINKING
705
OPTIMIZED EXECUTABLE

*FIG. 7*

# PROGRAMMATIC ACCESS TO THE WIDEST MODE FLOATING-POINT ARITHMETIC SUPPORTED BY A PROCESSOR

## TECHNICAL FIELD OF THE INVENTION

This invention relates generally to the compilation of computer code performing floating-point arithmetic operations on floating-point registers, and more particularly to enabling, via a new data type, access in source code to the full width of the floating point register representation in the underlying processor.

## BACKGROUND OF THE INVENTION

Source-level languages like C and C++ typically do not support constructs that enable access to low-level machine-instructions. Yet many instruction set architectures provide functionally useful machine instructions that cannot readily be accessed from standard source-level constructs.

Typically, programmers, and notably operating system developers, access the functionality afforded by these special (possibly privileged) machine-instructions from source programs by invoking subroutines coded in assembly language, where the machine instructions can be directly specified. This approach suffers from a significant performance drawback in that the overhead of a procedure call/return sequence must be incurred in order to execute the special machine instruction(s). Moreover, the assembly-coded machine instruction sequence cannot be optimized along with the invoking routine.

To overcome the performance limitation with the assembly routine invocation strategy, compilers known in the art, such as the Gnu C compiler ("gcc"), provide some rudimentary high-level language extensions to allow programmers to embed a restricted set of machine instructions directly into their source code. In fact, the 1990 American National Standard for Information Systems—Programming Language C (hereinafter referred to as the "ANSI Standard") recommends the "asm" keyword as a common extension (though not part of the standard) for embedding machine instructions into source code. The ANSI Standard specifies no details, however, with regard to how this keyword is to be used.

Current schemes that employ this strategy have drawbacks. For instance, gcc employs an arcane specification syntax. Moreover, the gcc optimizer does not have an innate knowledge of the semantics of embedded machine instructions and so the user is required to spell out the optimization restrictions. No semantics checks are performed by the compiler on the embedded instructions and for the most part they are simply "passed through" the compiler and written out to the target assembly file.

Other drawbacks of the inline assembly support in current compilers include:

(a) lack of functionality to allow the user to specify scheduling restrictions associated with embedded machine instructions. This functionality would be particularly advantageous with respect to privileged instructions.

(b) imposition of arbitrary restrictions on the kind of operands that may be specified for the embedded machine instructions, for example:
the compiler may require operands to be simple program variables (where permitting an arbitrary arithmetic expression as an operand would be more advantageous); and



the operands may be unable to refer to machine-specific resources in a syntactically natural manner.

(c) lack of functionality to allow the programmer to access the full range and precision of internal floating-point register representations when embedding floating-point instructions. This functionality would simplify high-precision or high-performance floating-point algorithms.

(d) imposition of restrictions on the ability to inline library procedures that include embedded machine instructions into contexts where such procedures are invoked, thereby curtailing program optimization effectiveness.

In addition, when only a selected subset of the machine opcodes are permitted to be embedded into user programs, it may be cumbersome in current compilers to extend the embedded assembly support for other machine opcodes. In particular, this may require careful modifications to many portions of the compiler source code. An extensible mechanism capable of extending embedded assembly support to other machine opcodes would reduce the number and complexity of source code modifications required.

It would therefore be highly advantageous to develop a compiler with a sophisticated capability for processing machine instructions embedded in high level source code. A "natural" specification syntax would be user friendly, while independent front-end validation would reduce the potential for many programmer errors. Further, it would be advantageous to implement an extensible compiler mechanism that processes source code containing embedded machine instructions where the mechanism is smoothly receptive to programmer-defined parameters indicating the nature and extent of compiler optimization permitted in a given case. A particularly useful application of such an improved compiler would be in coding machine-dependent "library" functions which would otherwise need to be largely written in assembly language and would therefore not be subject to effective compiler optimization, such as inlining.

In summary, there is a need for a compiler mechanism that allows machine instructions to be included in high-level program source code, where the translation and compiler optimization of such instructions offers the following advantageous features to overcome the above-described shortcomings of the current art:

a) a "natural" specification syntax for embedding low-level hardware machine instructions into high-level computer program source code.

b) a mechanism for the compiler front-end to perform syntax and semantic checks on the constructs used to embed machine instructions into program source code in an extensible and uniform manner, that is independent of the specific embedded machine instructions.

c) an extensible mechanism that minimizes the changes required in the compiler to support additional machine instructions.

d) a mechanism for the programmer to indicate the degree of instruction scheduling freedom that may be assumed by the compiler when optimizing high-level programs containing certain types of embedded machine instructions.

e) a mechanism to "inline" library functions containing embedded machine instructions into programs that invoke such library functions, in order to improve the run-time performance of such library function invocations, thereby optimizing overall program execution performance.

Such features would gain yet further advantage and utility in an environment where inline assembly support could gain access to the full width of the floating point registers in the target processor via specification of a corresponding data type in source code.

## SUMMARY OF THE INVENTION

These and other objects and features are achieved by one embodiment of the present invention which comprises the following:

1. A general syntax for embedding or "inlining" machine (assembly) instructions into source code. For each machine instruction that is a candidate for source-level inlining, an "intrinsic" (built-in subroutine) is defined. A function prototype is specified for each such intrinsic with enumerated data types used for instruction completers. The function prototype is of the following general form:

```
opcode_result=_Asm_opcode (opcode_argument_list[, serialization_constraint_specifier])
```

where _Asm_opcode is the name of the intrinsic function (with the "opcode" portion of the name replaced with the opcode mnemonic). Opcode completers, immediate source operands, and register source operands are specified as arguments to the intrinsic and the register target operand (if applicable) corresponds to the "return value" of the intrinsic.

The data types for register operands are defined to match the requirements of the machine instruction, with the compiler performing the necessary data type conversions on the source arguments and the return value of the "inline-assembly" intrinsics, in much the same way as for any user-defined prototyped function.

Thus, the specification syntax for embedding machine instructions in source code is quite "natural" in that it is very similar to the syntax used for an ordinary function call in most high-level languages (e.g. C, C++) and is subject to data type conversion rules applicable to ordinary function calls.

Further, the list of arguments for the machine opcode is followed by an optional instruction serialization_constraint_specifier. This feature provides the programmer a mechanism to restrict, through a parameter specified in source code, compiler optimization phases from re-ordering instructions across an embedded machine instruction.

This feature is highly advantageous in situations where embedded machine instructions may have implicit side-effects, needing to be honored as scheduling constraints by the compiler only in certain contexts known to the user. This ability to control optimizations is particularly useful for operating system programmers who have a need to embed privileged low-level "system" instructions into their source code.

Serialization_constraint_specifiers are predefined into several disparate categories. In application, the serialization_constraint_specifier associated with an embedded machine instruction is encoded as a bit-mask that specifies whether distinct categories of instructions may be re-ordered relative to the embedded machine instruction to dynamically execute either before or after that embedded machine instruction. The serialization constraint specifier is specified as an optional final argument to selected inline assembly intrinsics for which user-specified optimization control is desired. When this argument is omitted, a suitably conservative default value is assumed by the compiler.

2. A mechanism to translate the source-level inline assembly intrinsics from the source-code into a representation understood by the compiler back-end in a manner that is independent of the specific characteristics of the machine instruction being inlined.

The inline assembly intrinsics are "lowered" by the compiler front end into a built-in function-call understood by the code generation phase of the compiler. The code generator in turn expands the intrinsic into the corresponding machine instruction which is then subjected to low-level optimization.

An automated table-driven approach is used to facilitate both syntax and semantic checking of the inline assembly intrinsics as well as the translation of the intrinsics into actual machine instructions. The table contains one entry for each intrinsic, with the entry describing characteristics of that intrinsic, such as its name and the name and data types of the required opcode arguments and return value (if any), as well as other information relevant to translating the intrinsic into a low-level machine instruction.

The table is used to generate (1) a file that documents the intrinsics for user programmers (including their function prototypes) (2) a set of routines invoked by the compiler front-end to parse the supported inline assembly intrinsics and (3) a portion of the compiler back-end that translates the built-in function-call corresponding to each intrinsic into the appropriate machine instruction.

This table-driven approach requires very few, if any, changes to the compiler when extending source-level inline assembly capabilities to support the embedding of additional machine instructions. It is usually sufficient just to add a description of the new machine instructions to the table, re-generate the derived files, and re-build the compiler, so long as the low-level components of the compiler support the emission of the new machine instructions.

3. Where supported by a cross-module compiler optimization framework, a mechanism to capture the intermediate representation into a persistent format enables cross-module optimization of source-code containing embedded machine instructions. In particular, library routines with embedded machine instructions can themselves be "inlined" into the calling user functions, enabling more effective, context-sensitive optimization of such library routines, resulting in improved run-time performance of applications that invoke the library routines. This feature is highly advantageous, for instance, in the case of math library routines that typically need to manipulate aspects of the floating-point run-time environment through special machine instructions.

The inventive machine instruction inlining mechanism is also advantageously used in conjunction with a new data type which enables programmatic access to the widest mode floating-point arithmetic supported by the processor. As noted in the previous section, inline support in current compilers is generally unable to access the full range and precision of internal floating-point register representations when embedding floating-point instructions. Compiler implementations typically map source-level floating-point data types to fixed-width memory representations. The memory width determines the range and degree of precision to which real numbers can be represented. So, for example, an 8-byte floating-point value can represent a larger range of real numbers with greater precision than a corresponding 4-byte floating-point value. On some processors, however, floating-point registers may have a larger width than the natural width of source-level floating-point data types, allowing for intermediate floating-point results to be computed to a greater precision and numeric range; but this extended precision and range is not usually available to the user of a high-level language (such as C or C++).

In order to provide access to the full width of the floating point registers for either ordinary floating-point arithmetic or for inline assembly constructs involving floating-point operands, therefore, a new built-in data type is also disclosed herein, named "_fpreg" for the C programming language, corresponding to a floating point representation that is as wide as the floating-point registers of the underlying processor. Users may take advantage of the new data type in conjunction with the disclosed methods for the embedding of a machine instruction by using this data type for the parameters and/or return value of an intrinsic that maps to a floating-point machine instruction.

It is therefore a technical advantage of the present invention to enable a flexible, easy to understand language-compatible syntax for embedding or inlining machine instructions into source code.

It is a further technical advantage of the present invention to enable the compiler to perform semantic checks on the embedded machine instructions that are specified by invoking prototyped intrinsic routines, in much the same way that semantic checks are performed on calls to prototyped user routines.

It is a still further technical advantage of the present invention to enable inline assembly support to be extended to new machine instructions in a streamlined manner that greatly minimizes the need to modify compiler source code.

It is a yet further technical advantage of the present invention to enable user-controlled optimization of embedded low-level "system" machine instructions.

It is another technical advantage of the present invention to enable, where supported, cross-module optimizations, notably inlining, of library routines that contain embedded machine instructions.

Another technical advantage of the present invention is, when used in conjunction with the new _fpreg data type also disclosed herein, to support and facilitate reference to floating-point machine instructions in order to provide access to the full width of floating-point registers provided by a processor.

The foregoing has outlined rather broadly the features and technical advantages of the present invention in order that the detailed description of the invention that follows may be better understood. Additional features and advantages of the invention will be described hereinafter which form the subject of the claims of the invention. It should be appreciated by those skilled in the art that the conception and the specific embodiment disclosed may be readily utilized as a basis for modifying or designing other structures for carrying out the same purposes as the present invention. It should also be realized by those skilled in the art that such equivalent constructions do not depart from the spirit and scope of the invention as set forth in the appended claims.

## BRIEF DESCRIPTION OF THE DRAWINGS

For a more complete understanding of the present invention, and the advantages thereof, reference is now made to the following descriptions taken in conjunction with the accompanying drawings, in which:

FIG. **1** illustrates a typical compiler system in which the present invention may be enabled;

FIG. **2** illustrates the availability of an inline assembly intrinsic header file ("inline.h") $SH_2$ as a system header file containing intrinsics through which machine instructions may be inlined in accordance with the present invention;

FIG. **3** illustrates use of inline assembly descriptor table **301** to generate the "inline.h" system header file;

FIG. **4** illustrates use of inline assembly descriptor table **301** to generate a library of low-level object files available to assist translation of intrinsics from a high level intermediate representation of code to a low level intermediate representation thereof;

FIG. **5** illustrates use of inline assembly descriptor table **301** to create a library to assist front-end validation of intrinsics during the compilation thereof;

FIG. **6** illustrates the language-independent nature of file and library generation in accordance with the present invention; and

FIG. **7** illustrates an application of the present invention, where cross-module optimization and linking is supported, to enable performance critical library routines (such as math functions) to access low-level machine instructions and still allow such routines to be inlined into user applications.

## DESCRIPTION OF THE PREFERRED EMBODIMENTS

Turning first to FIG. **1**, a typical compilation process is illustrated upon which a preferred embodiment of the present invention may be enabled. The compiler transforms the source program on which it operates through several representations. Source code **101** is a representation created by the programmer. Front-end processing **102** transforms source code **101** into a high-level intermediate representation **103** used within the compiler. At this high-level intermediate stage, the associated high-level intermediate representation **103** is advantageously (although not mandatorily) passed through high-level optimizer **104** to translate the representation into a more efficient high-level intermediate representation. Code generator **105** then translates high-level intermediate representation **103** into low-level intermediate representation **106**, whereupon a low-level optimizer **107** translates low-level intermediate representation **106** into an efficient object code representation **108** that can be linked into a machine-executable program. It will be appreciated that some compilers are known in the art in which front-end and code generation stages **102** and **105** are combined (removing the need for high-level intermediate representation **103**), and in others, the code generation and low-level optimizing stages **105** and **107** are combined (removing the need for low-level intermediate representation **106**). Other compilers are known that add additional representations and translation stages. Within this basic framework, however, the present invention may be enabled on any analogous compiler performing substantially the steps described on FIG. **1**.

With reference now to FIG. **2**, the source level specification features of the present invention will now be discussed. It will be appreciated that consistent with earlier disclosure in the background and summary sections set forth above, the present invention provides a mechanism by which machine instructions may be embedded into source code to enable improved levels of smooth and efficient compilation thereof, advantageously responsive to selected optimization restrictions ordained by the programmer.

In a preferred embodiment herein, the invention is enabled on compilation of C source code. It will be appreciated, however, that use of the C programming language in this way is exemplary only, and that the invention may be enabled analogously on other high-level programming languages, such as C++ and FORTRAN, without departing from the spirit and scope of the invention.

Turning now to FIG. **2**, it should be first noted that blocks **202**, **203** and **204** are explanatory labels and not part of the

overall flow of information illustrated thereon. On FIG. 2, machine instructions are embedded or "inlined" into C source code $201_s$ by the user, through the use of "built-in" pseudo-function (or "intrinsic") calls. Generally, programs that make intrinsic calls may refer to and incorporate several types of files into source code 201., such as application header files $AH_1$–$AH_n$, or system header files $SH_1$–$SH_n$. In the exemplary use of C source code described herein, the present invention is enabled through inclusion of system header file $SH_2$ on FIG. 2, namely an inline assembly intrinsic header file (see label block 202). The file is specified as set forth below in detail so that, when included in source code $201_s$, the mechanism of the present invention will be enabled.

Note that as shown on FIG. 2, the inline assembly intrinsic header file $SH_2$ (named "inline.h" on FIG. 2 to be consistent with the convention on many UNIX®-based systems) typically contains declarations of symbolic constants and intrinsic function prototypes (possibly just as comments), and typically resides in a central location along with other system header files $SH_1$–$SH_n$. The "inline.h" system header file $SH_2$ may then be "included" by user programs that embed machine instructions into source code, enabling the use of _Asm_opcode intrinsics.

With reference to FIG. 3, it will be seen that a software tool 302 advantageously generates "inline.h" system header file $SH_2$ from an inline assembly descriptor table 301 at the time the compiler product is created. This table-driven approach is described in more detail later.

Returning to FIG. 2, when an inline assembly intrinsic header file specified in accordance with the present invention is included in a user program, processing continues to convert source code $201_s$ into object code $201_o$ in the manner illustrated on FIG. 1.

Syntax

The general syntax for the pseudo-function call/intrinsic call in C is as follows:

```
opcode_result=_Asm_opcode (<completer_list>, <operand_
  list>[,<serialization_constraint>]);
```

A unique built-in function name (denoted above as _Asm_opcode) is defined for each assembly instruction that can be generated using the inline assembly mechanism. The inline assembly instructions may be regarded as external functions for which implicit external declarations and function definitions are provided by the compiler. These intrinsics are not declared explicitly by the user programs. Moreover, the addresses of the inline assembly intrinsics may not be computed by a user program.

The "_Asm_opcode" name is recognized by the compiler and causes the corresponding assembly instruction to be generated. In general, a unique _Asm_opcode name is defined for each instruction that corresponds to a supported inline assembly instruction.

The first arguments to the _Asm_opcode intrinsic call, denoted as <completer_list> in the general syntax description, are symbolic constants for all the completers associated with the opcode. The inline assembly opcode completer arguments are followed by the instruction operands, denoted as <operand_list> in the general syntax description, which are specified in order according to protocols defined by particular instruction set architectures. Note that if an embedded machine instruction has no completers, then the <completer_list> and its following comma is omitted. Similarly, if the embedded machine instruction has no operands, then the <operand_list> and its preceding comma is omitted.

An operand that corresponds to a dedicated machine register (source or target) may be specified using an appropriate symbolic constant. Such symbolic constants are typically defined in the "inline.h" system header file discussed above with reference to FIG. 2. An immediate operand is specified as a simple integer constant and generally should be no larger than the corresponding instruction field width.

To be compatible, a source language expression having a scalar data type must be specified as the argument corresponding to a general purpose or floating-point register source operand. In particular, a general purpose register source operand must be specified using an argument expression that has an arithmetic data type (i.e. integral or floating-point type). An operand may nonetheless be of any type within this requirement for compatibility. For example, an operand may be a simple variable, or alternatively it may be an arithmetic expression including variables.

Typically, the compiler will convert the argument value for a general-purpose register source operand into an unsigned integer value as wide as the register width of the target machine (e.g. 32-bits or 64-bits).

Where a general-purpose register operand clearly corresponds to a memory address, an argument value having a pointer type may be required.

Any general purpose or floating-point register target operand value defined by an inline assembly instruction is treated as the return value of the _Asm_opcode pseudo-function call.

A general-purpose register target operand value is typically treated as an unsigned integer that is as wide as the register-width of the target architecture (e.g. 32-bits or 64-bits). Therefore, the pseudo-function return value will be subject to the normal type conversions associated with an ordinary call to a function that returns an unsigned integer value.

To avoid potential loss of precision when operating on floating-point values, however, floating-point register target and source operands of embedded machine instructions in a preferred embodiment are allowed to be as wide as the floating-point register-width of the target architecture. For architectures where the floating-point register-width exceeds the natural memory-width of standard floating-point data types (e.g. the "float" and "double" standard data types in the C language), the new "_fpreg" data type may be used to declare the arguments and return value of inline assembly intrinsics used to embed floating-point instructions into source code. As explained in more detail elsewhere in this disclosure, the "_fpreg" data type corresponds to a floating-point representation that is as wide as the floating-point registers of the target architecture.

The following examples illustrate the source code specification technique of the present invention as described immediately above:

i) For an "ADD" machine instruction of the form:

```
ADD r1=r2, r3
```

where r1, r2, and r3 correspond to 64-bit general-purpose machine registers, the function prototype for the inline intrinsic can be defined as follows:

```
UInt64 _Asm_ADD (UInt64 r2, UInt64 r3)
```

where "UInt64" corresponds to a 64-bit unsigned integer data-type.

The ADD machine instruction can then be embedded into a "C" source program as follows:

```
#include <inline.h>
int g1, g2, g3;                /* global integer variables */
main ( )
{
    g1 = _Asm_ADD (g2, g3);
}
```

ii) For a "LOAD" machine instruction of the form:

```
LOAD.<size>value=[mem_addr]
```

where

<size> is an opcode completer encoding the bit-size of the object being loaded which may be one of "b" (for byte or 8-bits), "hw" (for half-word or 16-bits) or "word" (for word or 32-bits),

"value" corresponds to a 32-bit general-purpose machine register whose value is to be set by the load instruction

"mem_addr" corresponds to a 32-bit memory address that specifies the starting location in memory of the object whose value is to be loaded into "value"

the function prototype for the inline intrinsic can be defined as follows:

```
UInt32 _Asm_LOAD (_Asm_size size, void*mem_addr)
```

where "UInt32" corresponds to a 32-bit unsigned integer data-type, "void*" is a generic pointer data type, and "_Asm_size" is a enumeration type that encodes one of 3 possible symbolic constants. For example, in the C language, _Asm_size may be defined as follows:

```
typedef enum {
  _b=1,
  _hw=2,
  _w=3
} _Asm_size;
```

Alternatively, _Asm_size may be defined to be a simple integer data type with pre-defined symbolic constant values for each legal LOAD opcode completer. Using language neutral "C" pre-processor directives,

```
#define _b (1)
#define _hw (2)
#define _w (3)
```

Note that the declarations associated with "_Asm_size" would be placed in the "inline.h" system header file, and would be read in by the compiler when parsing the source program.

The LOAD machine instruction can then be embedded into a "C" program thusly:

```
#include <inline.h>
int g;              /* global integer variable */
int *p;             /* global integer pointer variable */
main ( )
{
    g = _Asm_LOAD(_w, p);
}
```

Certain inline assembly opcodes, notably those that may be considered as privileged "system" opcodes, may optionally specify an additional argument that explicitly indicates the constraints that the compiler must honor with regard to instruction re-ordering. This optional "serialization constraint" argument is specified as an integer mask value. The integer mask value encodes what types of (data independent) instructions may be moved past the inline assembly opcode in either direction in a dynamic sense (i.e. before to after, or after to before) in the current function body. If omitted, the compiler will use a default serialization mask value. For the purposes of specifying serialization constraints in a preferred embodiment, the instruction opcodes may advantageously, but not mandatorily, be divided into the following categories:

1. Memory Opcodes: load and store instructions
2. ALU Opcodes: instructions with general-purpose register operands
3. Floating-Point Opcodes: instructions with floating-point register operands
4. System Opcodes: privileged "system" instructions
5. Branch: "basic block" boundary
6. Call: function invocation point

With respect to serialization constraints, an embedded machine instruction may act as a "fence" that prevents the scheduling of downstream instructions ahead of it, or a "fence" that prevents the scheduling of upstream instructions after it. Such constraints may be referred to as a "downward fence" and "upward fence" serialization constraint, respectively. Given this classification, the serialization constraints associated with an inline system opcode can be encoded as an integer value, which can be defined by ORing together an appropriate set of constant bit-masks. For a system opcode, this encoded serialization constraint value may be specified as an optional final argument of the _Asm_opcode intrinsic call. For example, for the C language, the bit-mask values may defined to be enumeration constants as follows:

```
typedef enum {
    _NO_FENCE          = 0x0,
    _UP_MEM_FENCE      = 0x1,
    _UP_ALU_FENCE      = 0x2,
    _UP_FLOP_FENCE     = 0x4,
    _UP_SYS_FENCE      = 0x8,
    _UP_CALL_FENCE     = 0x10,
    _UP_BR_FENCE       = 0x20,
    _DOWN_MEM_FENCE    = 0x100,
    _DOWN_ALU_FENCE    = 0x200,
    _DOWN_FLOP_FENCE   = 0x400,
    _DOWN_SYS_FENCE    = 0x800,
    _DOWN_CALL_FENCE   = 0x1000,
    _DOWN_BR_FENCE     = 0x2000
} _Asm_fence;
```

(Note: The _Asm_fence definition would advantageously be placed in the "inline.h" system header file.)

So, for example, to prevent the compiler from scheduling floating-point operations across an inlined system opcode that changes the default floating-point rounding mode, a programmer might use an integer mask formed as (_UP_FLOP_FENCE|_DOWN_FLOP_FENCE).

The _UP_BR_FENCE and _DOWN_BR_FENCE relate to "basic block" boundaries. (A basic block corresponds to the largest contiguous section of source code without any incoming or outgoing control transfers, excluding function calls.) Thus, a serialization constraint value formed by ORing together these two bit masks will prevent the compiler from scheduling the associated inlined system opcode outside of its original basic block.

Note that the compiler must automatically detect and honor any explicit data dependence constraints involving an inlined system opcode, independent of its associated serial-

ization mask value. So, for example, just because an inlined system opcode intrinsic call argument is defined by an integer add operation, it is not necessary to explicitly specify the _UP_ALU_FENCE bit-mask as part of the serialization constraint argument.

The serialization constraint integer mask value may be treated as an optional final argument to the inline system opcode intrinsic invocation. If this argument is omitted, the compiler may choose to use any reasonable default serialization mask value (e.g. 0x3D3D—full serialization with all other opcode categories except ALU operations). Note that if a system opcode instruction is constrained to be serialized with respect to another instruction, the compiler must not schedule the two instructions to execute concurrently.

To specify serialization constraints at an arbitrary point in a program, a placeholder inline assembly opcode intrinsic named _Asm_sched_fence may be used. This special intrinsic just accepts one argument that specifies the serialization mask value. The compiler will then honor the serialization constraints associated with this placeholder opcode, but omit the opcode from the final instruction stream.

The scope of the serialization constraints is limited to the function containing the inlined system opcode. By default, the compiler may assume that called functions do not specify any inlined system opcodes with serialization constraints. However, the _Asm_sched_fence intrinsic may be used to explicitly communicate serialization constraints at a call-site that is known to invoke a function that executes a serializing system instruction.

EXAMPLE

If a flush cache instruction ("FC" opcode) is a privileged machine instruction that is to be embedded into source code and one that should allow user-specified serialization constraints, the following inline assembly intrinsic may be defined:

```
void _Asm_FC ([serialization_constraint_specifier])
```

where the return type of the intrinsic is declared to be "void" to indicate that no data value is defined by the machine instruction.

Now the FC instruction may be embedded in a C program with serialization constraints that prevent the compiler from re-ordering memory instructions across the FC instruction as shown below:

```
#include <inline.h>
int g1, g2;    /* global integer variables */
main ( )
{
  g1 = 0;                    /* can't be moved after FC instruction */
  _Asm_FC(_UP_MEMORY_FENCE|DOWN_MEMORY_FENCE);
  g2 = 1;                    /* can't be moved before FC instruction */
}
```

Note that the _Asm_FC instruction specifies memory fence serialization constraints in both directions preventing the re-ordering of the stores to global variables g1 and g2 across the FC instruction.

Use of Table-driven Approach

A table-driven approach is advantageously used to help the compiler handle assembly intrinsic operations. The table contains one entry for each intrinsic, with the entry describing the characteristics of that intrinsic. In a preferred embodiment, although not mandatorily, those characteristics may be tabulated as follows:

(a) The name of the intrinsic

(b) A brief textual description of the intrinsic

(c) Names and types of the intrinsic arguments (if any)

(d) Name and type of the intrinsic return value (if any)

(e) With momentary reference back to FIG. **1**, additional information for code generator **105** to perform the translation from high level intermediate representation **103** to low level intermediate representation **106**

It will be appreciated that this table-driven approach enables the separation of the generation of the assembly intrinsic header file, parsing support library, and code generation utilities from the compiler's mainstream compilation processing. Any maintenance to the table may be made (such as adding to the list of supported inlined instruction) without affecting the compiler's primary processing functionality. This makes performing such maintenance easy and predictable. The table-driven approach is also user programming language independent, extending the versatility of the present invention.

On a more detailed level, at least three specific advantages are offered by this table-driven approach:

1. Header File Generation

The table facilitates generation of a file that documents intrinsics for user programmers, providing intrinsic function prototypes and brief descriptions. Using table elements (a), (b), (c) and (d) as itemized above, and with reference again to the preceding discussion accompanying FIG. **3**, a software tool **302** generates an "inline.h" system header $SH_2$ from inline assembly descriptor table **301**. Furthermore, "inline.h" system header $SH_2$ also defines and contains an enumerated set of symbolic constants, registers, completers, and so forth, that the programmer may use as legal operands to inline assembly intrinsic calls in the current program. Further, in cases where an operand is a numeric constant, "inline.h" system header $SH_2$ documents the range of legal values for the operand, which is checked by the compiler.

2. Parsing Library Generation

The table facilitates generation of part of a library that assists, with reference again now to FIG. **1**, front end processing ("FE") **102** in recognizing intrinsics specified by the programmer in source code **101**, validating first that the programmer has written such intrinsics legally, and then translating the intrinsics into high-level intermediate representation **103**.

Note that in accordance with the present invention, it would also be possible to generate intrinsic-related front-end processing directly. In a preferred embodiment, however, library functionality is used.

Table-driven front-end processing enables an advantageous feature of the present invention, namely the automatic syntax parsing and semantics checking of the user's inline assembly code by FE **102**. This feature validates that code containing embedded machine instructions is semantically correct when it is incorporated into source code **101** in the same way that a front end verifies that an ordinary function invocation is semantically correct. This frees other processing units of the compiler, such as code generator **105** and low level optimizer **107**, from the time-consuming task of error checking.

This front-end validation through reference to a partial library is enabled by generation of a header file as illustrated on FIGS. **5** and **6**. Turning first to FIG. **5**, in which it should again be noted that blocks **506**, **507** and **508** are explanatory

items and not part of the information flow, inline assembly descriptor table **301** provides elements (a), (c) and (d) as itemized above to software tool **501**. This information enables software tool **501** to generate language-independent inline assembly parser header file ("asmp.h"), which may then be included into corresponding source code "asmp.c" **503** and compiled **504** into corresponding object code "asmp.o" **505**. It will thus be seen from FIG. **5** that "asmp.o" **505** is a language-independent inline assembly parser library object file in a form suitable for assisting FE **102** on FIG. **1**.

With reference now to FIG. **6**, it will be seen that "inline.h" system header $SH_2$ provides legal intrinsics for a programmer to invoke from source code **601**. On FIG. **6**, exemplary illustration is made of C source code $601_c$, C++ source code $601_p$, and FORTRAN source code $601_f$, although the invention is not limited to these particular programming languages, and will be understood to be also enabled according to FIG. **6** on other programming languages. It will be noted that each of the illustrated source codes $601_c$, $601_p$ and $601_f$ have compiler operations and sequences **601**–**608** analogous to FIG. **1**. Further, "asmp.o" library object file **505**, being language independent, is universally available to C FE $602_c$, C++ FE $602_p$ and FORTRAN FE $602_f$ to assist in front-end error checking. Front end processing FE **602** does this checking by invoking utility functions defined in "asmp.o" library object file **505** to ensure that embedded machine instructions encountered in source code **601** are using the correct types and numbers of values. This checking is advantageously performed before actual code for embedded machine instructions is generated in high-level intermediate representation **603**.

In this way, it will be appreciated that various potential errors may be checked in a flexible, table-driven manner that is easily maintained by a programmer. For example, errors that may be checked include:

- whether the instruction being inlined is supported.
- whether the number of arguments passed is correct.
- whether the arguments passed are of the correct type.
- whether the values of numeric integer constant arguments, if any, are within the allowable range.
- whether the serialization constraint specifier is allowed for the specified instruction.

Furthermore, the table also allows the system to compute the default serialization mask for the specified instruction if one is needed but not supplied by the user.

### 3. Code Generation

The table **301** facilitates actual code generation (as shown on FIG. **1**) by assisting CG **105** in translation of high level intermediate representation ("HIL") **103** to low level intermediate representation ("LIL") **106**. Specifically, the table assists CG **105** in translating intrinsics previously incorporated into source code **101**. The table may also, when processed into a part of CG **105**, perform consistency checking to recognize certain cases of incorrect HIL **103** that were not caught by error checking in front end processing ("FE") **102**.

Note that according to the present invention, it would also be possible to generate a library of CG object files to assist CG **105** in processing intrinsics, similar to library **505** that assists FE **102**, as illustrated on FIG. **5**. Turning now to FIG. **4**, and again noting that blocks **405** and **406** are explanatory items and not part of the information flow, inline assembly descriptor assembly table **301** provides elements (c), (d) and (e) as itemized above to software tool **400**. Using this information, software tool **400** generates CG source file $401_1$, which in turn is compiled along with ordinary CG source files $401_2$–$401_n$ (blocks **402**) into CG object files $403_1$–$403_n$. Archiver **404** accumulates CG object files $403_1$–$403_n$ into CG library **407**.

In more detail now, the foregoing translation from HIL **103** to LIL **106** for intrinsics includes the following phases:

A. Generation of data structures

Automation at compiler-build time generates, for each possible intrinsic operation, a data structure that contains information on the types of the intrinsic arguments (if any) and the type of the return value (if any).

B. Consistency checking

At compiler-run time, a portion of CG that performs consistency checking on intrinsic operations can consult the appropriate data structure from A immediately above. This portion of CG does not need to be modified when a new intrinsic operation is added, unless the language in which the table **301** is written has changed.

C. Translation from HIL to LIL

Most intrinsic operations can be translated from HIL to LIL automatically, using information from the table. In a preferred embodiment, an escape mechanism is also advantageously provided so that an intrinsic operation that cannot be translated automatically can be flagged to be translated later by a hand-coded routine. The enablement of the escape mechanism does not affect automatic consistency checking.

The representation of an intrinsic invocation in HIL identifies the intrinsic operation and has a list of arguments; there may be an implicit return value. The representation of an intrinsic invocation in LIL identifies a low-level operation and has a list of arguments. The translation process must retrieve information from the HIL representation and build up the LIL representation. There are a number of aspects to this mapping:

i. The identity of the intrinsic operation in HIL may be expressed by one or more arguments in LIL. Information in element (e) in the inline assembly descriptor table set forth above is used to generate code expressing this identity in LIL.

ii. The implicit return value (if any) from HIL is expressed as an argument in LIL.

iii. Arguments of certain types in HIL must be translated to arguments of different types in LIL. The translation utility for any given argument type must be hand-coded, although the correct translation utility is invoked automatically by the translation process for the intrinsic operation.

iv. The serialization mask (if any) from HIL is a special attribute (not an argument) in LIL.

v. The LIL arguments must be emitted in the correct order. Information in element (e) in the inline assembly descriptor table as set forth above describes how to take the identity arguments from (i), the return value argument (if any) from (ii), and any other HIL arguments, and emit them into LIL in the correct order.

For each possible intrinsic operation, the tool run at compiler-build time creates a piece of CG that takes as input the HIL form of that intrinsic operation and generates the LIL form of that intrinsic operation.

In a preferred embodiment, the tool run at compiler-build time advantageously recognizes when two or more intrinsic operations are translated using the same algorithm, and

generates a single piece of code embodying that algorithm that can perform translations for all of those intrinsic operations. When this happens, information on the identity of the intrinsic operation described in (i) above is stored in the same data structures described in A further above, so that the translation code can handle the multiple intrinsic operations. In the preferred embodiment, translation algorithms for two intrinsic operations are considered "the same" if all of the following hold:

- The HIL forms of the operations have the same number of arguments of the same types in the same order.
- The HIL forms of the operations either both lack a return value or have the same return type.
- The identity information is expressed in the LIL forms of the operations using the same number of arguments of the same types.
- The LIL arguments for the operations occur in the same order.

In summary, within the internal program representations used by the compiler, the inlining of assembly instructions may be implemented as special calls in the HIL that the front end generates. Every assembly instruction supported by inlining is defined as part of this intermediate language.

When an inlined assembly instruction is encountered in the source, after performing error checking, the FE would emit, as part of the HIL, a call to the corresponding dedicated HIL routine.

The CG then replaces each such call in the HIL with the corresponding machine instruction in the LIL which is then subject to optimizations by the LLO, without violating any associated serialization constraint specifiers (as discussed above).

In addition to facilitating code generation from HIL to LIL, the table-driven approach advantageously assists code generation in other phases of the compiler. For example, and with reference again to FIG. **1**, the table could also be extended to generate part of HLO **104** or LLO **107** for manipulating assembly intrinsics (or to generate libraries to be used by HLO **104** or LLO **107**). This could be accomplished, for instance, by having the table provide semantic information on the intrinsics that indicates optimization freedom and optimization constraints. Although the greatest benefit comes from using the table for as many compiler stages as possible, this approach applies equally well to a situation in which only some of the compiler stages use the table—for example, where neither HLO **104** nor LLO **107** use the table.

Although the preferred embodiment does Library Generation and Partial Code Generator Generation (as described above) at compiler-build time, it would not be substantially different for FE **102**, CG **105**, or some library to consult the table (or some translated form of the table) at compiler-run time instead.

Furthermore, although this approach has been disclosed to apply to assembly intrinsics, it could equally well be applied to any set of operations where there is at least one compiler stage that takes a set of operations in a regular form and translates them into another form, where the translation process can occur in a straightforward and automated fashion.

Each time a new intrinsic operation needs to be added to the compiler, a new entry is added to the table of intrinsic operations. A compiler stage that relies on the table-driven approach usually need not be modified by hand in order to manipulate the new intrinsic operation (the exception is if the language in which the table itself is written has to be extended—for example, to accommodate a new argument type or a new return type; in such a case it is likely that compiler stages and automation that processes the table will have to be modified). Reducing the amount of code that must be written by hand makes it simpler and quicker to add support for new intrinsic operations, and reduces the possibility of error when adding new intrinsic operations.

A further advantageous feature enabled by the present invention is that key library routines may now access machine instruction-level code so as to optimize run-time performance. Performance-critical library routines (e.g. math or graphics library routines) often require access to low-level machine instructions to achieve maximum performance on modern processors. In the current art, they are typically hand-coded in assembly language.

As traditionally performed, hand-coding of assembly language has many drawbacks. It is inherently tedious, it requires detailed understanding of microarchitecture performance characteristics, it is difficult to do well and is error-prone, the resultant code is hard to maintain, and, to achieve optimal performance, the code requires rework for each new implementation of the target architecture.

In a preferred embodiment of the present invention, performance-critical library routines may now be coded in high-level languages, using embedded machine instructions as needed. Such routines may then be compiled into an object file format that is amenable to cross-module optimization and linking in conjunction with application code that invokes the library routines. Specifically, the library routines may be inlined at the call sites in the application program and optimized in the context of the surrounding code.

With reference to FIG. **7**, intrinsics defined in "inline.h" system header file $SH_2$ enable machine instructions to be embedded, for example, in math library routine source code $702_s$. This "mathlib" source code $702_s$ is then compiled in accordance with the present invention into equivalent object code $702_o$. Meanwhile, source code $701_s$ wishing to invoke the functionality of "mathlib" is compiled into object code $701_o$ in the traditional manner employed for cross-module optimization. Cross-module optimization and linking resources **704** then combine the two object codes $701_o$ and $702_o$ to create optimized executable code **705**.

In FIG. **7**, it should be noted that the math library is merely used as an example. There are other analogous high-performance libraries for which the present invention brings programming advantages, e.g., for graphics, multimedia, etc.

In addition to easing the programming burden on library writers, the ability to embed machine instructions into source code spares the library writers from having to re-structure low-level hand-coded assembly routines for each implementation of the target architecture.

Floating Point ("_fpreg") Data Type

The description of a preferred embodiment has so far centered on the inventive mechanism disclosed herein for inlining machine instructions into the compilation and optimization of source code. It will be appreciated that this mechanism will often be called upon to compile objects that include floating-point data types. A new data type is also disclosed herein, named "_fpreg" in the C programming language, which allows general programmatic access (including via the inventive machine instruction inlining mechanism) to the widest mode floating-point arithmetic supported by the processor. This data type corresponds to a floating-point representation that is as wide as the floating-point registers of the underlying processor. It will be understood that although discussion of the inventive data type herein centers on "_fpreg" as named for the C programming

language, the concepts and advantages of the inventive data type are applicable in other programming languages via corresponding data types given their own names.

A precondition to fully enabling the "_fpreg" data type is that the target processor must of course be able to support memory access instructions that can transfer data between its floating-point registers and memory without loss of range or precision.

Depending on the characteristics of the underlying processor, the "_fpreg" data type may be defined as a data type that either requires "active" or "passive" conversion. The distinction here is whether instructions are emitted when converting a value of "_fpreg" data type to or from a value of another floating-point data type. In an active conversion, a machine instruction would be needed to effect the conversion whereas in a passive conversion, no machine instruction would be needed. In either case, the memory representation of an object of "_fpreg" data type is defined to be large enough to accommodate the full width of the floating-point registers of the underlying processor.

The type promotion rules of the programming language are advantageously extended to accommodate the _fpreg data type in a natural way. For example, for the C programming language, it is useful to assert that binary operations involving this type shall be subject to the following promotion rules:

1. First, if either operand has type _fpreg, the other operand is converted to _fpreg.
2. Otherwise, if either operand has type long double, the other operand is converted to long double.
3. Otherwise, if either operand has type double, the other operand is converted to double.

Note that in setting the foregoing exemplary promotion rules, it is assumed that the _fpreg data type which corresponds to the full floating-point register width of the target processor has greater range and precision than the long double data type. If this is not the case, then the first two rules may need to be swapped in sequence.

Note also that in general, assuming type _fpreg has greater range and/or precision than type long double, it may be that the result of computations involving _fpreg values cannot be represented precisely as a value of type long double. The behavior of the type conversion from _fpreg to long double (or to any other source-level floating-point type) must therefore be accounted for. A preferred embodiment employs a similar rule to that used for conversions from double to float: If the value being converted is out of range, the behavior is undefined; and if the value cannot be represented exactly, the result is either the nearest higher or the nearest lower representable value.

It will be further appreciated that the application and availability of the _fpreg data type is not required to be universal within the programming language. Depending on processor architecture and programmer needs, it is possible to limit availability of the _fpreg data type to only a subset of the operations that may be applied to other floating-point types.

To illustrate general programming use of this new data type, consider the following C source program that computes a floating-point 'dot-product' (a·b+c):

```
double a, b, c, d;
main ( )
{
    d = (a * b) + c;
}
```

where the global variable d is assigned the result of the dot-product. For this example, according to the standard "usual arithmetic conversion rule" of the C programming language, the floating-point multiplication and addition expressions will be evaluated in the "double" data type using double precision floating-point arithmetic instructions. However, in order to exploit greater precision afforded by a processor with floating-point registers whose width exceeds that of the standard double data type, the _fpreg data type may alternatively be used as shown below:

```
double a, b, c, d;
main ( )
{
    d = ((_fpreg) a * b) + c;
}
```

Note here that the variable "a" of type double is "typecast" into an _fpreg value. Hence, based on the previously mentioned extension to the usual arithmetic conversion rule, the variables "a", "b", and "c" of "double" type are converted (either passively or actively) into "_fpreg" type values and both the multiplication and addition operations will operate in the maximum floating-point precision corresponding to the full width of the underlying floating-point registers. In particular, the intermediate maximum precision product of "a" and "b" will not need to be rounded prior to being summed with "c". The net result is that a more accurate dot-product value will be computed and round-off errors are limited to the final assignment to the variable "d".

Applying the foregoing features and advantages of the _fpreg data type to the inventive mechanism disclosed herein for inlining machine instructions, it will be seen that the parameters and return values of intrinsics specified in accordance with that mechanism may be declared to be of this data type when such intrinsics correspond to floating point instructions.

For example, in order to allow source-level embedding of a floating-point fused-multiply add instruction:

```
fma fr4=fr1, fr2, fr3
```

that sums the product of the values contained in 2 floating-point register source operands (fr1 and fr2) with the value contained in another floating-point register source operand (fr3), and writes the result to a floating-point register (fr4), the following inline assembly intrinsic can be defined:

```
fr4=_fpreg _Asm_fma (_fpreg fr1, _fpreg fr2, _fpreg fr3)
```

Now, following the general programmatic example used above, this intrinsic can be used to compute a floating-point "dot-product" (a·b+c) in a C source program as follows:

```
double a, b, c, d;
main ( )
{
    d = _Asm_fma (a, b, c);
}
```

where d is assigned the result of the floating-point computation ((a*b)+c)

Note that the arguments to _Asm_fma (a, b, and c) are implicitly converted from type double to type _fpreg when invoking the intrinsic, and that the intrinsic return value of type _fpreg is implicitly converted to type double for assignment to d. As discussed above, if type _fpreg has greater range and/or precision than type double, it may be that the result of the intrinsic operation (or indeed any other expression of type _fpreg) cannot be represented precisely as a value of type double. The behavior of the type conversion from _fpreg to double (or to any other source-level floating-point type, such as float) must therefore be accounted for. In a preferred embodiment, a similar rule is employed to that used for conversions from double to float: If the value being converted is out of range, the behavior is undefined; and if the value cannot be represented exactly, the result is either the nearest higher or nearest lower representable value.

If the result of the dot-product were to be used in a subsequent floating-point operation, it would be possible to minimize loss of precision by carrying out that operation in type _fpreg as follows:

```
double a, b, c, d, e, f, g;
main ( )
{
_fpreg x, y;
    x = _Asm_fma (a, b, c);
    y = _Asm_fma (e, f, g);
    d = x + y;
}
```

Note that the results of the two dot-products are stored in variables of type _fpreg; the results are summed (still in type _fpreg), and this final sum is then converted to type double for assignment to d. This should produce a more precise result than storing the dot-product results in variables of type double before summing them. Also, note that the standard binary operator '+' is being applied to values of type _fpreg to produce an _fpreg result (which, as previously stated, must be converted to type double for assignment to d).

Conclusion

It will be further understood that the present invention may be embodied in software executable on a general purpose computer including a processing unit accessing a computer-readable storage medium, a memory, and a plurality of I/O devices.

Although the present invention and its advantages have been described in detail, it should be understood that various changes, substitutions and alterations can be made herein without departing from the spirit and scope of the invention as defined by the appended claims.

We claim:

1. A method for enabling access in the C programming language to the widest mode floating-point arithmetic supported by a processor, wherein the processor can transfer data between memory and its floating-point registers without loss of range or precision, the method comprising the steps of:

(a) defining "_fpreg" as a floating-point data type having a memory representation that has range and precision at least as great as said floating-point registers;

(b) declaring selected operands to be of the floating-point data type, said selected operands specified in C source code on which floating-point arithmetic is to be performed, said selected operands including arguments and return values of inline assembly intrinsics used to embed floating-point instructions into the source code;

(c) compiling the source code into corresponding object code;

(d) executing the object code;

(e) performing said floating-point arithmetic, said step (e) including the substep of transferring data between memory representations corresponding to said selected operands and said floating-point registers; and

(f) converting data types according to type promotion rules when said floating-point arithmetic combines operands having different data types including _fpreg, the promotion rules including hierarchical rules regarding conversion between data types to create data type compatibility between a pair of source operands, the promotion rules including, in order of precedence:

(1) if one source operand is of data type _fpreg, convert the other source operand to data type _fpreg;

(2) if one source operand is of data type long double, convert the other source operand to data type long double; and

(3) if one source operand is of data type double, convert the other source operand to data type double;

and wherein rule (2) takes precedence over rule (1) if the memory representation of a long double data type has greater range and precision than said floating-point registers.

* * * * *

```
Set     Items   Description
S1     727783   PREPROCESSOR? ? OR PRE()PROCESSOR? ? OR PREPROCESSER? ? OR
              PRE()PROCESSER? ? OR PRAGMA OR DIRECTIVE OR METAPROGRAMM??? OR
               COMPILER? ? OR INTERPRETER? ? OR PRECOMPILER? ? OR PROBE
S2   18491418   INSERT??? OR ADD??????? OR EMBED???? OR INCLUD??? OR INCLU-
              S??? OR INTEGRAT??? OR IMPLEMENT????? OR AUGMENT????? OR UPDA-
              T? OR UP()(DATE? OR DATING? OR GRAD?) OR UPGRAD? OR GENERAT???
               OR EXTEND??? OR EXTENS??? OR INCORPORAT???
S3    9905600   CONSTRUCT? ? OR FUNCTION? ? OR DATA()TYPE? ? OR LANGUAGE()-
              ELEMENT? ? OR OPERAND? ? OR OPERATOR? ? OR OPERATION? ? OR CO-
              NTROL()STRUCTURE? ? OR STRUCTURE()DEFINITION? ? OR SYMBOL? ? -
              OR BOOLEAN? ? OR (HIGH()LEVEL OR USER()FRIENDLY)()LANGUAGE? ?
              OR ROUTINE? ?
S4    2627324   EAS??? OR FACILITAT??? OR SIMPLIF???????
S5       7501   S1 AND S2 AND S3 AND S4
S6      26196   ASL OR ADVANCED(2W)CONFIGURATION(2W)POWER(2W)INTERFACE OR -
              ACPI OR AML
S7          5   S5 AND S6
S8       1288   AU=(QURESHI S? OR QURESHI, S?)
S9          0   S8 AND S6
S10    707004   S2(5N)S3
S11      1639   (S10 AND S1 AND S4) NOT S7
S12       239   S11 AND S1/TI
S13       196   S12 AND (PY<2003 OR PD<20021023)
S14       139   RD  (unique items)
S15       494   (EXTEND??? OR EXTENS??? OR LENGTHEN??? OR ENLARG???)(5N)DA-
              TA()TYPE? ?
S16         3   (S15 AND S1 AND S4) NOT (S7 OR S14)
? show files
File   2:INSPEC 1898-2006/Aug W1
         (c) 2006 Institution of Electrical Engineers
File   6:NTIS 1964-2006/Jul W5
         (c) 2006 NTIS, Intl Cpyrght All Rights Res
File   8:Ei Compendex(R) 1970-2006/Aug W1
         (c) 2006 Elsevier Eng.  Info. Inc.
File  34:SciSearch(R) Cited Ref Sci 1990-2006/Aug W1
         (c) 2006 The Thomson Corp
File  35:Dissertation Abs Online 1861-2006/Jun
         (c) 2006 ProQuest Info&Learning
File  56:Computer and Information Systems Abstracts 1966-2006/Jul
         (c) 2006 CSA.
File  57:Electronics & Communications Abstracts 1966-2006/Jul
         (c) 2006 CSA.
File  60:ANTE: Abstracts in New Tech & Engineer 1966-2006/Jul
         (c) 2006 CSA.
File  65:Inside Conferences 1993-2006/Aug 11
         (c) 2006 BLDSC all rts. reserv.
File  94:JICST-EPlus 1985-2006/May W1
         (c)2006 Japan Science and Tech Corp(JST)
File  95:TEME-Technology & Management 1989-2006/Aug W1
         (c) 2006 FIZ TECHNIK
File  99:Wilson Appl. Sci & Tech Abs 1983-2006/Jul
         (c) 2006 The HW Wilson Co.
File 111:TGG Natl.Newspaper Index(SM) 1979-2006/Aug 01
         (c) 2006 The Gale Group
File 144:Pascal 1973-2006/Jul W4
         (c) 2006 INIST/CNRS
File 256:TecInfoSource 82-2006/Nov
         (c) 2006 Info.Sources Inc
File 434:SciSearch(R) Cited Ref Sci 1974-1989/Dec
         (c) 2006 The Thomson Corp
```

BIBLIOGRAPHIC NPL/
NPL INVENTOR

```
376749  CONSTRUCT? ?
        5345332  FUNCTION? ?
        7779592  DATA
        5429827  TYPE? ?
          23859  DATA(W)TYPE? ?
         720926  LANGUAGE
        3624501  ELEMENT? ?
            443  LANGUAGE(W)ELEMENT? ?
           7364  OPERAND? ?
         584509  OPERATOR? ?
        2574468  OPERATION? ?
        6174094  CONTROL
        7842873  STRUCTURE? ?
          24009  CONTROL(W)STRUCTURE? ?
        6249928  STRUCTURE
         444716  DEFINITION? ?
            521  STRUCTURE(W)DEFINITION? ?
         156463  SYMBOL? ?
          63071  BOOLEAN? ?
        9013836  HIGH
        3104209  LEVEL
         247577  HIGH(W)LEVEL
         708397  USER
          81834  FRIENDLY
          33649  USER(W)FRIENDLY
         868841  LANGUAGE? ?
          37476  (HIGH(W)LEVEL OR USER(W)FRIENDLY)(W)LANGUAGE? ?
         254781  ROUTINE? ?
          33871  SUBROUTINE? ?
        4599817  SUB
         254781  ROUTINE? ?
            486  SUB(W)ROUTINE? ?
          78727  MACRO? ?
         509859  LIMITATION? ?
         807155  RULE? ?
     S3 9905600  CONSTRUCT? ? OR FUNCTION? ? OR DATA()TYPE? ? OR
                 LANGUAGE()ELEMENT? ? OR OPERAND? ? OR OPERATOR? ? OR
                 OPERATION? ? OR CONTROL()STRUCTURE? ? OR
                 STRUCTURE()DEFINITION? ? OR SYMBOL? ? OR BOOLEAN? ? OR
                 (HIGH()LEVEL OR USER()FRIENDLY)()LANGUAGE? ? OR ROUTINE?
                 ? OR SUBROUTINE? ? OR SUB()ROUTINE? ? OR MACRO? ? OR
                 LIMITATION? ? OR RULE? ?
```

14/9/127 (Item 18 from file: 35)
DIALOG(R)File 35:Dissertation Abs Online


816598 ORDER NO: AAD83-15921
**THE DESIGN OF A FAST COMPILER - COMPILER FOR PROGRAMMING LANGUAGES WITH LL(1) SYNTAX**

Author: BRYANT, BARRETT RICHARD
Degree: PH.D.
Year: 1983
Corporate Source/Institution: NORTHWESTERN UNIVERSITY (0163)
Source: VOLUME 44/03-B OF DISSERTATION ABSTRACTS INTERNATIONAL.
PAGE 848. 199 PAGES
Descriptors: COMPUTER SCIENCE
Descriptor Codes: 0984

The automatic production of syntax-organized **compilers** using a modified pushdown automaton to perform LL(1) parsing instead of recursive descent is investigated. The technique is similar to transition matrices, allowing very fast **compilers** , but considerably smaller. Unlike traditional LL(1) parsing, strings of input symbols are checked by one state instead of several, thereby reducing the number of states required. An integer value indicating how many input symbols to advance in a state transition is used instead of a Boolean value simply indicating if a symbol is consumed. Multiple error entries are also eliminated.

In addition to generating the syntax analyzer automatically, semantics must be automatically incorporated into the **compiler** from some formal specification. The programming language PASCAL is investigated as a metalanguage for defining **compiler** -oriented semantics, **including** the use of semantic **macros** in PASCAL to define similar features for a variety of programming languages as a means of automating static semantics and intermediate code expressible in PASCAL to define dynamic semantics. The intermediate language is defined to serve for a variety of source languages and target machines. PASCAL is used to completely define the semantics of itself and considered for defining FORTRAN and ADA. Because PASCAL is a programming language instead of a mathematical formal notation, it is **easier** to use a semantic metalanguage but its main advantage is that a PASCAL definition of a language constitutes a **compiler** for that language.

Algorithms were developed and implemented to generate LL(1) **compiler** tables from language specifications and to use the generated tables to compile source programs and it is shown how the pushdown automaton parsing technique can be extended to LL(k) parsing, for k > 1. A complete specification of PASCAL was used to generate a PASCAL **compiler** and syntactic specifications of FORTRAN and ADA were developed to generate parsing tables for these languages. These parsing tables are much smaller than parsing tables generated by other methods. The PASCAL **compiler** generated is comparable to a hand-coded recursive-descent **compiler** in terms of compilation time and contains significantly less source code. Therefore, an advantage is gained in both time and space in **compilers** generated by this technique.

14/9/102 (Item 20 from file: 8)
DIALOG(R)File 8:Ei Compendex(R)


00706644 E.I. Monthly No: EI7804023886 E.I. Yearly No: EI78015160
Title: **MARGOT: A MACRO -BASED GENERATOR OF COMMAND LANGUAGE** INTERPRETERS .
Author: Cashman, Paul M.; Myszewski, Mathew J.
Corporate Source: Mass Comput Assoc, Inc, Wakefield
Source: Proc of the Digital Equip Comput Users Soc, v 3 n 4 1977, Boston, Mass, May 31-Jun 3 1977 Publ by Digital Equip Corp, Maynard, Mass, 1977 p 1131-1144
Publication Year: 1977
Language: ENGLISH
Journal Announcement: 7804
Abstract: MARGOT is a system consisting of a set of metalanguage operators which can be used to describe the syntax and semantics of command languages. The **operators** are **implemented** as **macros** which expand to produce operation codes for a " MARGOT machine. " The latter is implemented as an **interpreter** written in PDP-11 assembler language. MARGOT is designed to be a powerful, problem-oriented, **easy** -to-learn language which corresponds naturally to BNF and allows a command's syntax and semantics to be associated **easily** . MARGOT includes facilities for definition of syntactic and semantic constructs, iteration, and specification of input and syntactic choice. The assembly-time and run-time actions of the MARGOT system are presented. Experience of MARGOT users is discussed, and MARGOT's advantages and shortcomings are analyzed. 21 refs.
Descriptors: *COMPUTER OPERATING SYSTEMS--*Program **Interpreters**
Classification Codes:
723 (Computer Software)
72 (COMPUTERS & DATA PROCESSING)

14/9/79 **(Item 16 from file: 6)**
DIALOG(R)File 6:NTIS


0570627 NTIS Accession Number: AD-A029 055/1/XAB
**The Augment** Precompiler . I. **User Information**
(Technical summary rept)
Crary, F. D.
Wisconsin Univ Madison Mathematics Research Center
Corp. Source Codes: 221200
Report No.: MRC-TSR-1469
Apr 76 149p
Document Type: Translation
Journal Announcement: GRAI7622
Supersedes report dated Dec 74, AD-A006 542. See also AD-A020 198.
Order this product from NTIS by: phone at 1-800-553-NTIS (U.S. customers); (703)605-6000 (other countries); fax at (703)321-8547; and email at orders@ntis.fedworld.gov. NTIS is located at 5285 Port Royal Road, Springfield, VA, 22161, USA.
NTIS Prices: PC A07/MF A01
Contract No.: DAAG29-75-C-0024
AUGMENT allows the **easy** use of packages **implementing** nonstandard **data types** and **operations** . This report describes the use of the **precompiler** , the preparation of supporting packages implementing nonstandard packages. Technical documentation is contained in a forthcoming companion report. (Author)
Descriptors: ***Compilers** ; *Computer programming; Fortran; Programming manuals; Mathematical logic; Programming languages; Machine translation; High level languages; User needs; Digital computers
Identifiers: ***Precompilers** ; **Augment precompilers** ; **Nonstandard data type** ; UNIVAC 1100 computers; Translations; NTISDODXA
Section Headings: 62B (Computers, Control, and Information Theory--Computer Software)

14/9/78 (Item 15 from file: 6)
DIALOG(R)File 6:NTIS


0572766 NTIS Accession Number: AD-A029 312/6/XAB
**Evaluation of HAL/S Language Compilability Using SAMSO's Compiler Writing System (CWS)**
(Final rept. Mar 75-Feb 76)
Feliciano, M. ; Anderson, H. D. ; Bond, J. W.
Aerospace Corp El Segundo Calif Information Processing Div
Corp. Source Codes: 409525
Report No.: TR-0076(6803)-1; SAMSO-TR-76-137
20 Aug 76 70p
Journal Announcement: GRAI7623
Order this product from NTIS by: phone at 1-800-553-NTIS (U.S. customers); (703)605-6000 (other countries); fax at (703)321-8547; and email at orders@ntis.fedworld.gov. NTIS is located at 5285 Port Royal Road, Springfield, VA, 22161, USA.
NTIS Prices: PC A04/MF A01
Contract No.: F04701-75-C-0076
NASA/Langley is engaged in a program to develop an adaptable guidance and control software concept for spacecraft such as shuttle-launched payloads. It is envisioned that this flight software be written in a higher-order language, such as HAL/S, to **facilitate** changes or additions. To make this adaptable software transferable to various onboard computers, a **compiler** writing system capability is necessary. A joint program with the Air Force Space and Missile Systems Organization was initaiated to determine if the **Compiler** Writing System (CWS) owned by the Air Force could be utilized for this purpose. The present study explores the feasibility of **including** the HAL/S language **constructs** in CWS and the effort required to **implement** these **constructs** . This will determine the compilability of HAL/S using CWS and permit NASA/Langley to identify the HAL/S constructs desired for their applications. The study consisted of comparing the implementation of the Space Programming Language using CWS with the requirements for the implementation of HAL/S. It is the conclusion of the study that CWS already contains many of the language features of HAL/S and that it can be expanded for compiling part or all of HAL/S. It is assumed that persons reading and evaluating this report have a basic familiarity with (1) the principles of **compiler** construction and operation, and (2) the logical structure and applications characteristics of HAL/S and SPL. (Author)
Descriptors: ***Compilers** ; *Programming languages; *High level languages; *Computer programming; Spacecraft; Space shuttles; Onboard; Computer programs; Computer program documentation; Payload
Identifiers: ***Compiler** writing system; HAL/S programming language; Computer software; CDC-6600 computers; Space programming language; NTISDODXA
Section Headings: 62B (Computers, Control, and Information Theory--Computer Software)

14/9/3 (Item 3 from file: 2)
DIALOG(R)File 2:INSPEC


08467179 INSPEC Abstract Number: C2003-01-7440-055
**Title: Development of preprocessor program for articulated total body**
Author(s): Lee Dong Jea; Son Kwon; Jeon Kyunam; Choi Kyunghyun
Author Affiliation: Pusan Nat. Univ., South Korea
Conference Title: ICCAS 2002. International Conference on Control, Automation and Systems p.2701-4
Publisher: Inst. Control, Autom. & Syst. Eng, Taejon, South Korea
Publication Date: 2001 Country of Publication: South Korea CD-ROM pp.
Material Identity Number: XX-2001-01846
Conference Title: Proceedings of 2001 International Conference on Control, Automation and Systems (16th Korea Automatic Control Conference)
Conference Sponsor: Korea Res. Found.; Korea Sci. & Eng. Found.; Korea Nat. Tourism Organ.; Korean Federation of Sci. & Technol. Soc
Conference Date: 17-21 Oct. 2001 Conference Location: Jeju Island, South Korea
Language: Korean Document Type: Conference Paper (PA)
Treatment: Applications (A); Practical (P)
Abstract: Computer simulations are widely used to analyze passenger safety in traffic accidents. ATB (articulated total body) is a computer simulation model developed to predict gross human body response to such dynamic environments as vehicle crashes and pilot ejections. ATB, whose code is open, has high flexibility and application capability that users can **easily insert** defined modules and **functions** . ATB is, however, inconvenient as it was coded in FORTRAN and it needs a formated input file. Moreover, it takes much time to make input files and to modify coding errors. The study aims to increase user friendliness by adding a **preprocessor** program, WINATB (WINdow ATB), to the conventional ATB. WINATB programmed in Visual C++ and OpenGL uses ATB IV as a dynamic solver. The **preprocessor** helps users prepare input files through graphic interface and dialog box and an additional postprocessor presents graphical presentation of simulated results. In the case of a frontal crash, the results obtained by WINATB and MADYMO are compared to validate the effectiveness of WINATB. (7 Refs)
Subfile: C
Descriptors: accidents; digital simulation; road vehicles; safety
Identifiers: **preprocessor** program; articulated total body; computer simulations; passenger safety; traffic accidents; gross human body response ; dynamic environments; vehicle crashes; pilot ejections; user friendliness ; WINATB; OpenGL; Visual C++; ATB IV; dynamic solver; graphic interface; dialog box; graphical presentation; frontal crash; MADYMO
Class Codes: C7440 (Civil and mechanical engineering computing)

```
Set     Items   Description
S1    3219344   (INSERT??? OR ADD??????? OR EMBED???? OR INCLUD??? OR INCL-
            US??? OR INTEGRAT??? OR IMPLEMENT????? OR AUGMENT????? OR UPD-
            AT? OR UP()(DATE? OR DATING? OR GRAD?) OR UPGRAD? OR GENERAT?-
            ?? OR EXTEND??? OR EXTENS??? OR INCORPORAT???)(5N)(CONSTRUCT?
            ? OR FUNCTION
S2     657350   PREPROCESSOR? ? OR PRE()PROCESSOR? ? OR PREPROCESSER? ? OR
            PRE()PROCESSER? ? OR PRAGMA OR DIRECTIVE OR METAPROGRAMM??? OR
             COMPILER? ? OR INTERPRETER? ? OR PRECOMPILER? ? OR PROBE
S3   10936287   EAS??? OR FACILITAT??? OR SIMPLIF???????
S4      45929   S1 AND S2 AND S3
S5      22762   ASL OR ADVANCED(2W)CONFIGURATION(2W)POWER(2W)INTERFACE OR -
            ACPI OR AML
S6        147   S4 AND S5
S7        101   RD  (unique items)
S8         60   S7 AND (PY<2003 OR PD<20021023)
S9       9880   S1(100N)S2(100N)S3
S10         9   (S9 AND S5) NOT S8
S11       681   (S9 AND (PREPROCESSOR? ? OR PRE()PROCESSOR? ? OR PREPROCES-
            SER? ? OR PRE()PROCESSER? ? OR PRAGMA OR METAPROGRAMM??? OR C-
            OMPILER? ? OR INTERPRETER? ? OR PRECOMPILER? ? OR ADVANCED(2W-
            )CONFIGURATION(2W)POWER(2W)INTERFACE OR ACPI)/TI) NOT (S8 OR -
            S10)
S12      1464   S1(20N)S2(20N)S3
S13       198   (S12 AND (PREPROCESSOR? ? OR PRE()PROCESSOR? ? OR PREPROCE-
            SSER? ? OR PRE()PROCESSER? ? OR PRAGMA OR METAPROGRAMM??? OR -
            COMPILER? ? OR INTERPRETER? ? OR PRECOMPILER? ? OR ADVANCED(2-
            W)CONFIGURATION(2W)POWER(2W)INTERFACE OR ACPI)/TI) NOT (S8 OR
            S10)
S14       547   S1(10N)S2(10N)S3
S15        90   (S14 AND (PREPROCESSOR? ? OR PRE()PROCESSOR? ? OR PREPROCE-
            SSER? ? OR PRE()PROCESSER? ? OR PRAGMA OR METAPROGRAMM??? OR -
            COMPILER? ? OR INTERPRETER? ? OR PRECOMPILER? ? OR ADVANCED(2-
            W)CONFIGURATION(2W)POWER(2W)INTERFACE OR ACPI)/TI) NOT (S8 OR
            S10)
S16        55   RD  (unique items)
S17        55   S16 AND (PY<2003 OR PD<20021023)
? show files
File 275:Gale Group Computer DB(TM) 1983-2006/Aug 11
         (c) 2006 The Gale Group
File  47:Gale Group Magazine DB(TM) 1959-2006/Aug 11
         (c) 2006 The Gale group
File  16:Gale Group PROMT(R) 1990-2006/Aug 11
         (c) 2006 The Gale Group
File 624:McGraw-Hill Publications 1985-2006/Aug 14
         (c) 2006 McGraw-Hill Co. Inc
File 484:Periodical Abs Plustext 1986-2006/Aug W1
         (c) 2006 ProQuest
File 613:PR Newswire 1999-2006/Aug 14
         (c) 2006 PR Newswire Association Inc
File 813:PR Newswire 1987-1999/Apr 30
         (c) 1999 PR Newswire Association Inc
File 239:Mathsci 1940-2006/Oct
         (c) 2006 American Mathematical Society
File 370:Science 1996-1999/Jul W3
         (c) 1999 AAAS
File 696:DIALOG Telecom. Newsletters 1995-2006/Aug 12
         (c) 2006 Dialog
File 621:Gale Group New Prod.Annou.(R) 1985-2006/Aug 11
         (c) 2006 The Gale Group
File 674:Computer News Fulltext 1989-2006/Jul W5
```

FULL TEXT NPL

(c) 2006 IDG Communications
File 88:Gale Group Business A.R.T.S. 1976-2006/Aug 02
(c) 2006 The Gale Group
File 369:New Scientist 1994-2006/Jul W3
(c) 2006 Reed Business Information Ltd.
File 160:Gale Group PROMT(R) 1972-1989
(c) 1999 The Gale Group
File 635:Business Dateline(R) 1985-2006/Aug 12
(c) 2006 ProQuest Info&Learning
File 15:ABI/Inform(R) 1971-2006/Aug 14
(c) 2006 ProQuest Info&Learning
File 9:Business & Industry(R) Jul/1994-2006/Aug 11
(c) 2006 The Gale Group
File 13:BAMP 2006/Aug W1
(c) 2006 The Gale Group
File 810:Business Wire 1986-1999/Feb 28
(c) 1999 Business Wire
File 610:Business Wire 1999-2006/Aug 14
(c) 2006 Business Wire.
File 647:CMP Computer Fulltext 1988-2006/Sep W4
(c) 2006 CMP Media, LLC
File 98:General Sci Abs 1984-2005/Jan
(c) 2006 The HW Wilson Co.
File 148:Gale Group Trade & Industry DB 1976-2006/Aug 11
(c)2006 The Gale Group
File 634:San Jose Mercury Jun 1985-2006/Aug 12
(c) 2006 San Jose Mercury News
File 636:Gale Group Newsletter DB(TM) 1987-2006/Aug 11
(c) 2006 The Gale Group
?

```
562920  INSERT???
      23105565  ADD???????
        793646  EMBED????
      25813235  INCLUD???
        722723  INCLUS???
       7258925  INTEGRAT???
       4474273  IMPLEMENT?????
        328491  AUGMENT?????
       3428694  UPDAT?
      19807682  UP
       5427356  DATE?
        213453  DATING?
       3032119  GRAD?
         22882  UP(W)((DATE? OR DATING?) OR GRAD?)
       2162517  UPGRAD?
       6957223  GENERAT???
       3773898  EXTEND???
       3278454  EXTENS???
       2701296  INCORPORAT???
        604013  CONSTRUCT? ?
       4060748  FUNCTION? ?
      12547911  DATA
      25259351  TYPE? ?
         39840  DATA(W)TYPE? ?
       1780140  LANGUAGE
       2151103  ELEMENT? ?
           618  LANGUAGE(W)ELEMENT? ?
         19539  OPERAND? ?
       2923136  OPERATOR? ?
      10537611  OPERATION? ?
       7000203  CONTROL
       3931300  STRUCTURE? ?
         10937  CONTROL(W)STRUCTURE? ?
       2789534  STRUCTURE
       1165887  DEFINITION? ?
           533  STRUCTURE(W)DEFINITION? ?
       1642783  SYMBOL? ?
         51102  BOOLEAN? ?
      13893239  HIGH
       6666845  LEVEL
        643851  HIGH(W)LEVEL
       3216617  USER
       1005123  FRIENDLY
        259030  USER(W)FRIENDLY
       2097199  LANGUAGE? ?
         12908  (HIGH(W)LEVEL OR USER(W)FRIENDLY)(W)LANGUAGE? ?
        558874  ROUTINE? ?
         16654  SUBROUTINE? ?
        961414  SUB
        558874  ROUTINE? ?
           691  SUB(W)ROUTINE? ?
        190159  MACRO? ?
       1130031  LIMITATION? ?
       3432441  RULE? ?
    S1 3219344  (INSERT??? OR ADD??????? OR EMBED???? OR INCLUD??? OR
                INCLUS??? OR INTEGRAT??? OR IMPLEMENT????? OR
                AUGMENT????? OR UPDAT? OR UP()(DATE? OR DATING? OR GRAD?)
                OR UPGRAD? OR GENERAT??? OR EXTEND??? OR EXTENS??? OR
                INCORPORAT???)(5N)(CONSTRUCT? ? OR FUNCTION? ? OR
                DATA()TYPE? ? OR LANGUAGE()ELEMENT? ? OR OPERAND? ? OR
                OPERATOR? ? OR OPERATION? ? OR CONTROL()STRUCTURE? ? OR
```

```
STRUCTURE()DEFINITION? ? OR SYMBOL? ? OR BOOLEAN? ? OR
(HIGH()LEVEL OR USER()FRIENDLY)()LANGUAGE? ? OR ROUTINE?
? OR SUBROUTINE? ? OR SUB()ROUTINE? ? OR MACRO? ? OR
LIMITATION? ? OR RULE? ?)
```

```
Set     Items   Description
S1     475835   (INSERT??? OR ADD??????? OR EMBED???? OR INCLUD??? OR INCL-
             US??? OR INTEGRAT??? OR IMPLEMENT????? OR AUGMENT????? OR UPD-
             AT? OR UP()(DATE? OR DATING? OR GRAD?) OR UPGRAD? OR GENERAT?-
             ?? OR EXTEND??? OR EXTENS??? OR INCORPORAT???)(5N)(CONSTRUCT?
             ? OR FUNCTION
S2     163373   PREPROCESSOR? ? OR PRE()PROCESSOR? ? OR PREPROCESSER? ? OR
             PRE()PROCESSER? ? OR PRAGMA OR DIRECTIVE OR METAPROGRAMM??? OR
              COMPILER? ? OR INTERPRETER? ? OR PRECOMPILER? ? OR PROBE
S3    1111542   EAS??? OR FACILITAT??? OR SIMPLIF???????
S4      57634   S1 AND S2 AND S3
S5       5439   ASL OR ADVANCED(2W)CONFIGURATION(2W)POWER(2W)INTERFACE OR -
             ACPI OR AML
S6       1248   S4 AND S5
S7          3   S6 AND IC=(G06F-009/45 OR G06F-001/26)
S8       3152   S1(100N)S2(100N)S3
S9         37   (S8 AND S5) NOT S7
S10        21   S9 NOT PD=(2002023:20060724)
S11        61   (S8 AND (PREPROCESSOR? ? OR PRE()PROCESSOR? ? OR PREPROCES-
             SER? ? OR PRE()PROCESSER? ? OR PRAGMA OR METAPROGRAMM??? OR C-
             OMPILER? ? OR INTERPRETER? ? OR PRECOMPILER? ? OR ADVANCED(2W-
             )CONFIGURATION(2W)POWER(2W)INTERFACE OR ACPI)/TI) NOT (S7 OR -
             S10)
S12        36   S11 NOT PD=(2002023:20060724)
? show files
File 348:EUROPEAN PATENTS 1978-2006/ 200632
         (c) 2006 European Patent Office
File 349:PCT FULLTEXT 1979-2006/UB=20060803,UT=20060727
         (c) 2006 WIPO/Univentio
?
```

FULL TEXT
PATENT

```
 591995  INSERT???
1204855  ADD???????
 143035  EMBED????
1519591  INCLUD???
 163854  INCLUS???
 413634  INTEGRAT???
 375521  IMPLEMENT?????
 186165  AUGMENT?????
 138060  UPDAT?
1149023  UP
2217433  DATE?
   1954  DATING?
 309005  GRAD?
   1966  UP(W)((DATE? OR DATING?) OR GRAD?)
  23915  UPGRAD?
 905125  GENERAT???
 855565  EXTEND???
 300431  EXTENS???
 801812  INCORPORAT???
 135209  CONSTRUCT? ?
 941848  FUNCTION? ?
 730144  DATA
1373622  TYPE? ?
  14423  DATA(W)TYPE? ?
  77518  LANGUAGE
1130270  ELEMENT? ?
    284  LANGUAGE(W)ELEMENT? ?
   7809  OPERAND? ?
 202127  OPERATOR? ?
1010580  OPERATION? ?
1038063  CONTROL
 980343  STRUCTURE? ?
   3857  CONTROL(W)STRUCTURE? ?
 896211  STRUCTURE
 220416  DEFINITION? ?
    545  STRUCTURE(W)DEFINITION? ?
 168520  SYMBOL? ?
  10004  BOOLEAN? ?
1220936  HIGH
 741660  LEVEL
 107931  HIGH(W)LEVEL
 393225  USER
  26249  FRIENDLY
  10630  USER(W)FRIENDLY
  81562  LANGUAGE? ?
   2158  (HIGH(W)LEVEL OR USER(W)FRIENDLY)(W)LANGUAGE? ?
 132823  ROUTINE? ?
  14107  SUBROUTINE? ?
 452466  SUB
 132823  ROUTINE? ?
   2456  SUB(W)ROUTINE? ?
  29147  MACRO? ?
 338188  LIMITATION? ?
 176353  RULE? ?
S1 475835  (INSERT??? OR ADD??????? OR EMBED???? OR INCLUD??? OR
           INCLUS??? OR INTEGRAT??? OR IMPLEMENT????? OR
           AUGMENT????? OR UPDAT? OR UP()(DATE? OR DATING? OR GRAD?)
           OR UPGRAD? OR GENERAT??? OR EXTEND??? OR EXTENS??? OR
           INCORPORAT???)(5N)(CONSTRUCT? ? OR FUNCTION? ? OR
           DATA()TYPE? ? OR LANGUAGE()ELEMENT? ? OR OPERAND? ? OR
           OPERATOR? ? OR OPERATION? ? OR CONTROL()STRUCTURE? ? OR
```

```
STRUCTURE()DEFINITION? ? OR SYMBOL? ? OR BOOLEAN? ? OR
(HIGH()LEVEL OR USER()FRIENDLY)()LANGUAGE? ? OR ROUTINE?
? OR SUBROUTINE? ? OR SUB()ROUTINE? ? OR MACRO? ? OR
LIMITATION? ? OR RULE? ?)
```